VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Nasazení algoritmů pro dělení rozsáhlých grafů na cluster pomocí MPI

Deployment algorithms for partitioning large graphs on a cluster using MPI

2014 Bc. Lukáš Kawulok

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Lukáš Kawulok**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Nasazení algoritmů pro dělení rozsáhlých grafů na cluster pomocí MPI.
Deployment Algorithms for Partitioning Large Graphs on a Cluster using MPI.

Zásady pro vypracování:

Tato diplomová práce má za cíl prostudovat oblast dělení rozsáhlých grafů a jejich aplikace. V implementační části budou jednotlivé algoritmy (například dělení grafů metodou random walk) nasazeny v paralelní podobě na HPC clustru.

1. Prostudovat a vypracovat přehled algoritmů pro dělení rozsáhlých grafů.
2. Zaměřit se na metodu random walk pro dělení grafů, změny ohodnocení hran pomocí RW, detekce shluků pomocí RW.
3. Implementovat dělení grafů pomocí MPI na cluster.
4. Provést experimenty s různými datovými kolekcemi.
5. Porovnat a shodnotit výsledky experimentů.

Seznam doporučené odborné literatury:

[1] Kwok, T., Lau, L. Finding small sparse cuts locally by random walk. pp. 1-9, 2012
[2] Fjallstrom, P. Algorithms for graph partitioning: A survey. Science 3, pp. 1-34, 1998
[3] Elsner, U. Graph partitioning - a survey. MONARCH Dokumenten und Publikationsservice Germany, pp. 1-58, 2005

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Mgr. Pavla Dráždilová, Ph.D.**

Datum zadání: 01.09.2013

Datum odevzdání: 07.05.2014

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 6. května 2014 .....

Rád bych na tomto místě poděkoval Mgr. Pavle Dráždilové, Ph.D., za vytrvalost při častých dotazech a za předání mnoha zkušeností a znalostí.

Abstrakt

Tato práce je zaměřena na metodu random walk a její aplikaci při dělení grafů. V práci jsou uvedeny základní pojmy z oblasti teorie grafů týkající se dělení grafu. Dále je prostor věnován různým metodám shlukování rozsáhlých grafů. Samotný random walk je naimplementován dvěma způsoby. A to jak sekvenčně, tak paralelně s využitím MPI knihovny. Naimplementované algoritmy byly testovány z různých pohledů a kvalita výsledného shlukování byla měřena pomoci několika metod.

Klíčová slova: náhodná procházka, shlukovací algoritmy, MPI knihovna, diplomová práce

Abstract

This work is focused on method called random walk. It contains explanation of basic definitions in field of graphs concerning graph partitioning. Further attention is then given to some methods of clustering of big graphs. Random walk method will be also implemented in two ways - serial and paralel. Paralel implementation will be based on MPI library. Implemented algorithms were tested from different points of view and quality of clustering was also measured using several measuring methods.

Keywords: random walk, clustering algorithms, MPI library, master thesis

Seznam použitých zkratek a symbolů

MPI	–	Message Passing Interface
NS	–	Neighbourhood Similarity
GDF	–	Geographic Data Format
KL	–	Kernighan-Lin
RW	–	Random Walk
HPC	–	High Performance Computing
RAM	–	Random-Access Memory
VS	–	Výskyty ve sledech
SI	–	Silhouette Index

Obsah

Seznam tabulek

Seznam obrázků

Seznam výpisů zdrojového kódu

Seznam algoritmů

1 Úvod

Grafy jsou v dnešní době, ať si to uvědomujeme, či nikoliv, stále více součástí našeho každodenního života. Můžeme je nalézt třeba v pravidelných autobusových linkách, telefonních sítích, zobrazovacích technikách, atd. Vzhledem k tomu, že se některé grafy dovedou rozrůst do obřích rozměrů, je velice vhodné si v rámci analýzy graf pomocí shlukování rozdělit na menší části. Příkladem takového grafu může být například grafické znázornění sociální sítě Facebook. Shlukování využívá přirozené provázání vrcholů, které spolu nějakým způsobem souvisí. Na základě toho, že jsme z různých grafů získali tyto shluky, můžeme určit teoretické organizace v sociálních sítích, či vyhledat parazitující buňky v lidském těle.

Díky tomu, že se rychlost našich počítačů stále zvyšuje, je v dnešní době dělení grafů čím dál tím rychlejší. Tato diplomová práce se věnuje algoritmům pro dělení grafů. Existuje celá řada metod, které tuto problematiku řeší. Důvodem vysokého množství algoritmů je nejednoznačnost v tom, který z nich je nejlepší. Výběr metody závisí na požadavcích, které jsou na algoritmus kladeny. Některé z metod se vyznačují vysokou rychlostí, jiné zase přesností. Z vysokého množství různých metod byla vybrána metoda *random walk* pro podrobnější popsání, a seznámení se s touto metodou.

I přesto, že se rychlost počítačů stále zvyšuje, je zde stále možnost urychlení výpočtů. Tohoto urychlení se dosáhne pomocí rozdělení práce na více procesorů, ke kterému dochází pomoci knihovny MPI, která zajišťuje komunikaci mezi těmito počítači. Pro využívání MPI knihovny, která nám umožňuje využívání více výpočetních uzlů, je nutno nalézt algoritmus *random walk*, který se dá výpočetně rozdělit na výpočetní uzly. Cílem je tedy nalézt algoritmus, který lze buďto datově, či časově rozdělit. Pokud tento algoritmus není nalezen, pokoušíme se vhodný algoritmus navrhnout a následně implementovat. Tento algoritmus by měl být navrhnut jak pro sekvenční výpočet, tak i pro více výpočetních uzlů s očekáváním odpovídajícího zrychlení. Samotný čas komunikace mezi výpočetními uzly by neměl být příliš náročný, aby nezpomalil algoritmus natolik, že bude pomalejší než jeho sekvenční verze. V případě, že by k tomuto došlo, algoritmus nebyl správně navrhnut.

Navržené algoritmy jsou testovány na základě různých kriterií tak, aby bylo možno ověřit, zda implementovaný program pracuje korektně, a zda je získané dělení grafů vyhovující. Kvalita rozdělení grafů je v experimentech ověřována podle vypočítaných kvalitativních parametrů. Na závěr jsou provedeny testy pro ověření rychlosti algoritmu při využití MPI knihovny.

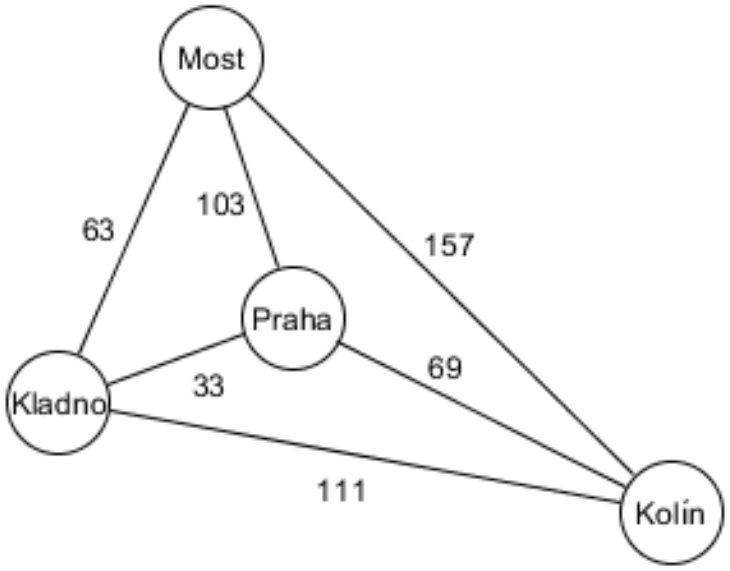
2 Základní teoretické informace

Před zabýváním se samotným dělením grafů je nutno objasnit základní pojmy, které se váží ke grafům. Uvedeny jsou základní definice těchto pojmů. Prvním z nich je teorie grafů, zabývající se vlastnostmi struktury, která je tvořena vrcholy propojenými hranami. Tyto struktury nazýváme grafy. Nejčastěji se můžeme setkat s grafickým znázorněním v podobě přímkami propojených bodů (viz.[1]). Tato kapitola je také zaměřena na knihovnu MPI, která slouží ke komunikaci mezi výpočetními uzly pro urychlení procesu dělení grafů.

2.1 Graf

Každý člověk si jistě dokáže nějaký graf představit. Matematické grafy, jako například sloupcový, či koláčový [2] však není zájmem této práce. Graf představuje abstrakci reálného světa, kde objekty zájmu znázorňujeme pomocí bodů, a vztahy mezi těmito body značí hrany. Na obrázku níže můžeme vidět jednoduchý příklad, kdy vrcholy značí města a hrany cesty mezi nimi. Hodnota hran udává vzdálenost mezi městy (viz Obr. 1).

Obrázek 1: Vzdálenost mezi městy



2.2 Základní pojmy týkající se grafu

V této části práce jsou uvedeny definice tykající se grafu, které jsou použité dále při dělení grafu. Nejdříve je však potřeba uvést definici 2.1 samotného grafu. Tyto informace jsou čerpány z [2].

Definice 2.1 *Graf je definován jako dvojice dvou množin vrcholů (V) a hran (H). Zároveň také jako zobrazení* $\epsilon : E \to V^2$*, kde je každé hraně přiřazena dvojice vrcholů.*

- $G = (V, H, \epsilon)$ *nebo G = (V,H)*
- $\epsilon(e) = (x, y)\epsilon V^2$

Definice 2.2 *Úplný graf je takový graf, ve kterém jsou každé dva uzly propojeny hranou.*

Definice 2.3 *Cesta v grafu je posloupnost vrcholů a hran, které vedou z vrcholu A do vrcholu B. Všechny vrcholy v cestě jsou rozdílné.*

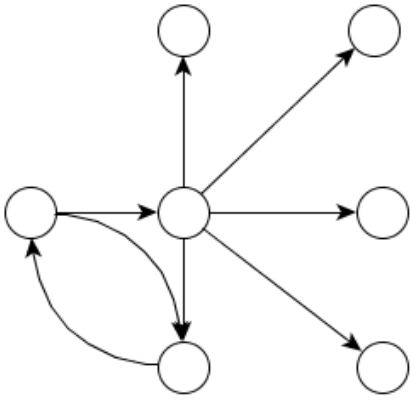
- *př: (* v_0*,* e_1*,* v_1*,...,* e_t*,* v_t*) kde i = 1,2,...,t a* $e_i = (v_{i-1}, v_i)$ *je prvkem E(G).*

Definice 2.4 *Graf G je souvislý, pokud pro každé body A a B v grafu G vede cesta z bodu A do bodu B.*

Definice 2.5 *Cyklus je posloupnost vrcholů a hran v orientovaném grafu, kde první a poslední vrchol je stejný a ostatní jsou různé. Nejkratší cyklus má minimální délku 3.*

Definice 2.6 *Orientovaný graf je dvojice (V, E), kde E je podmnožina kartézského součinu* $V \times V$*. Orientovaná hrana e má tvar (x, y) což znamená, že vychází z x a končí v y Obr. 2.*

Obrázek 2: Ukázka orientovaného grafu



Definice 2.7 *Pokud hrany mají k sobě přiřazené hodnoty, pak se hovoří o ohodnocené hraně. (Ukázku je uvedena na počátku této kapitoly) Obr. 1.*

Definice 2.8 *Stupeň vrcholu (někdy též valence vrcholu) označuje počet hran, které do daného vrcholu zasahují. Stupeň vrcholu u se značí* $deg(u)$*.*

Definice 2.9 *Shluk je skupina prvků určitého grafu, které si jsou navzájem více podobné než prvky v jiném shluku. Každý vrchol grafu může náležet pouze jednomu shluku.*

- Orientace hran

  Orientovaný graf - hrany jsou brány, jako uspořádané dvojice vrcholů.

  Neorientovaný graf - hrany jsou dvouprvkové množiny vrcholů, každá hrana značí cestu tam i zpět.

- Ohodnocení hran

  Ohodnocený graf - každá hrana v grafu má přiřazenou váhu.

  Neohodnocený graf - hrany nemají přiřazené váhy, nebo se může také říci, že mají všechny stejnou váhu.

- Souvislosti grafu

  Souvislý graf - takový graf, ve kterém existuje cesta mezi každými dvěma vrcholy.

  Nesouvislý graf - takový graf, kde neexistuje cesta mezi jakýmikoliv dvěma vrcholy.

2.3 Reprezentace grafu v počítači

Reprezentovat graf lze více způsoby. Některé jsou paměťově méně náročné, jiné zase přehlednější. Každá reprezentace má něco do sebe, tato práce se však zabývá pouze několika z nich.

2.3.1 Maticová reprezentace

Do kategorie maticové reprezentace jsou řazeny všechny reprezentace, které nějakým způsobem využívají matici. Lze si to představit jako dvou dimenzionální pole prvků x_{ij}, kde i a j značí vrcholy, a jejich hrany jsou reprezentovány hodnotou proměnné x_{ij}. Z této definice plyne, že na rozdíl od jiných reprezentací je právě tato velice přehledná. Pro rozsáhlé grafy je však těžce použitelná, a to z důvodu prostorové náročnosti.

2.3.1.1 Matice sousednosti Matice sousednosti se značí A. U neorientovaných grafů každý prvek a_{ij} nabývá dvou různých hodnot, buď 0 nebo 1. V případě, že $a_{ij} = 1$, pak mezi vrcholy v_i a v_j hrana existuje. V případě hodnoty 0, hrana neexistuje. U orientovaných grafů hodnotu 1 nabývá prvek a_{ij} pouze tehdy, pokud hrana směřuje od vrcholu v_i k vrcholu v_j.

- Neorientovaný graf:

  $a_{ij} = 1$: Hrana existuje mezi v_i a v_j

  $a_{ij} = 0$: Mezi vrcholy v_i a v_j neexistuje hrana

- Orientovaný graf:

  $a_{ij} = 1$: Hrana směřuje od v_i do v_j

  $a_{ij} = 0$: Neexistuje taková hrana směřující od v_i do v_j

2.3.1.2 Matice cen Další možností maticové reprezentace je matice cen C . Ta se používá, pokud jsou hrany grafu ohodnoceny w_{ij}. V takovém případě se prvky matice c_{ij} rovnají ohodnocení hrany w_{ij}, mezi vrcholy v_i a v_j. Pokud se jedná o orientovaný graf, pak velikost ohodnocené hrany w_{ij} nabývá pouze c_{ij}, pokud hrana směřuje od vrcholu v_i do v_j.

- Neorientovaný ohodnocený graf:

  $c_{ij} = w_{ij}$: Hrana existuje mezi v_i a v_j

  $c_{ij} = 0$: Mezi vrcholy v_i a v_j neexistuje hrana

- Orientovaný ohodnocený graf:

  $c_{ij} = w_{ij}$: Hrana směřuje od v_i do v_j

  $c_{ij} = 0$: Neexistuje taková hrana směřující od v_i do v_j

V případě neorientovaného grafu lze brzy rozpoznat, že matice je symetrická podle hlavní diagonály. Tuto vlastnost lze využít ke snížení paměťové náročnosti na polovinu. Maticová orientace je zbytečně paměťově náročná, pokud je v grafu málo hran. V tomto případě pak matice zabírá veliké množství paměti, a obsahuje málo informací. Tento jev se nazývá řídká matice.

2.3.2 Řídké matice

Jedním ze základních problémů, na který lze narazit při práci s velkými grafy, je řídkost výskytu hran mezi vrcholy. Vhodným řešením je upravit graf do formátu, ve kterém nebudeme uchovávat informace o neexistujících hranách, ale pouze o těch existujících. Takový způsob reprezentace grafu vede ke zefektivnění výpočtu a snížení velikosti.

2.3.3 Seznamová reprezentace

Jak je již zmíněno výše, maticová reprezentace ač je přehledná, tak je i velice paměťově náročná. Alternativou je seznamová reprezentace, jejíž paměťová náročnost je závislá na počtu hran. Tato reprezentace využívá dvojice, kde každá z nich značí jednu hranu. Jejími prvky jsou vrcholy, které tato hrana spojuje. V případě ohodnoceného grafu jsou třetím členem ohodnocení hrany. Pokud chceme tímto způsobem reprezentovat orientované hrany, pak první člen značí vrchol, ze kterého hrana vychází, a druhý do kterého směřuje. V případě obousměrné hrany je nutné reprezentovat oba tyto směry vlastními hranami. Níže lze vidět příklady grafu se čtyřmi vrcholy.

- Neorientovaný ohodnocený graf $G(V, E, w)$:

  $(v_1, v_2, 2); (v_2, v_3, 1); (v_3, v_4, 3); (v_4, v_1, 5);$

- Orientovaný ohodnocený graf $G(V, Ew)$:

  $(v_1, v_2, 2); (v_2, v_3, 1); (v_3, v_4, 3); (v_4, v_1, 5); (v_4, v_3, 1); (v_5, v_5, 6);$

2.4 MPI

Díky neustále se zvyšujícímu výkonu počítačů se dělení rozsáhlých grafů stává snažší úlohou [3]. Jednou z možností, která se nabízí, je (obrazně) práce více lidí na jednom díle. Což znamená, využití více jader na jednom procesoru, nebo druhou alternativou je využití více procesů. Právě druhou možností se tato práce dále zabývá. Bohužel tato myšlenka neznamená automatické navýšení výpočtu o množství procesorů, vznikají totiž následující problémy:

2.4.1 Škálovatelnost

Tento problém vychází z toho, že některé algoritmy nelze převést na paralelní, a proto musí být prováděny sekvenčně. I tyto algoritmy lze zrychlit, ale táto práce se tímto nezabývá. Více informací o této problematice je k dispozici zde: [4].

2.4.2 Rozložení zátěže

Při využívaní více procesorů je důležité, aby každý z procesorů byl využíván stejně. To znamená, aby se nestávalo, že jeden proces počítá, a druhý nemá co dělat. Situaci komplikuje i fakt, že jednotlivá jádra spolu musí komunikovat a synchronizovat se. Z toho důvodu, může kvůli kritickým sekcím jeden procesor čekat dlouhou dobu na ten druhý.

2.4.3 Komunikace

Přenášení dat a synchronizace nás přivádí k dalšímu problému. Při "real life"procesech, je nezbytné využívat komunikaci mezi procesy, která je velice časově náročná. Jedna se především o spouštění komunikace. Posílání stovek bytu je přibližně stejně náročné, jako posílání jednoho bytu.

2.4.4 Knihovna HPC cluster

HPC cluster nebo také cluster znamená propojení několika počítačů mezi sebou, které se však jeví jako jedna výpočetní jednotka. Jednotlivé počítače v tomto případě nazýváme uzly. Každý uzel má svůj vlastní operační systém a své vlastní procesy. Uzly se dělí na vedoucí a výpočetní. Vedoucí uzel provádí řízení a plánování úloh běžících na clusteru. Zajišťuje také komunikaci clusteru s uživatelem. Výpočetní uzel provádí naplánované úlohy, které poskytuje plánovač úloh. HPC Cluster může být nainstalovaný na více operačních systémech, tato diplomová práce byla vyvíjena na operačním systému Windows a v jazyce C#. Byl tedy využit .NET Framework od společnosti Microsoft a knihovna MPI.NET. Více zde [17].

2.4.5 Knihovna MPI.NET

K využívaní více počítačů pro dělení grafu byla vybrána knihovna MPI.NET, která slouží ke komunikaci uzlů mezi sebou. Data lze posílat jednak z uzlu na uzel, nebo také posílat na více uzlů současně. Z pohledu referenčního modelu ISO/OSI je MPI na vrstvě 5 (relační

vrstva). MPI je nezávislá na programovacích jazycích. Tato práce využila MPI pro jazyk C#.

Vytvoření aplikace s použitím MPI.NET není nijak složité. Do projektu, ve kterém se má MPI používat, stačí přidat referenci na MPI knihovnu, a vytvořit MPI prostředí pomocí MPI.Enviroment. Samotné používání není obtížné. Je potřeba však vždy počítat s tím, že daný kód bude využíván všemi uzly. Z tohoto důvodu musíme zabezpečit, aby do kritických částí kódu přistupoval jen ten proces, který chceme (Klasický příklad jsou vedoucí a výpočetní uzly). Jednoduchou ukázku programu lze vidět na zdrojovém kódu 1. Tento program vypisuje za každý proces své číslo, a celkový počet spuštěných procesů v rámci jednoho programu. Pokud tedy program spouštíme tímto příkazem *./mpiexec -n 6 newprogram.exe*, pak získáme stejný výsledek, kde *mpiexec* říká, že se program má spustit pomocí MPI knihovny. Parametr *n* uvádí, kolik procesů má spustit a *newprogram.exe* je program, který se má spustit. Více funkcí je k nalezení v [18].

```
static void Main(string[] args)
{
  using (new MPI.Environment(ref args))
     {
          Console.WriteLine("Jsem proces cislo : " + Communicator.world.Rank + " z " +
               MPI.Environment.Size + ".");
     }
  }

  // Vypis z konzole
  Jsem proces cislo 1 z 6.
  Jsem proces cislo 2 z 6.
  Jsem proces cislo 3 z 6.
  Jsem proces cislo 4 z 6.
  Jsem proces cislo 5 z 6.
  Jsem proces cislo 6 z 6.
```

Výpis 1: Vytvoření MPI prostředí a příklad výpisu čísla procesu [17]

Dále práce ukazuje, jak posílat data mezi jednotlivými uzly. Popisuje základní posílání, a to od jednoho uzlu k jinému specifickému. Poté jsou uvedeny některé funkce, které slouží k posílání informací ke všem uzlům najedou, nebo sesbíraní dat od všech uzlů, a následné zaslání vedoucímu uzlu. Těchto funkcí je více, a lze je nalézt na tomto odkazu: [18].

2.4.6 Komunikace mezi dvěma procesy

Pro komunikaci mezi dvěma procesy slouží metody *Send* a *Recieve*, které lze najít ve třídě *Communicator.world*. V parametrech metod jsou kromě samotných přenášených dat, také identifikace příjemce či odesilatele a hodnota, která rozlišuje zprávy mezi stejnými procesy. Ve výpisu níže (2) je odesilatel proces 0, který posílá data procesu číslo 1. Tato zpráva je identifikovatelná číslem 3.

```
double[] pole;
  Intracommunicator com = Communicator.world;

  if (com.Randk == 0)
```

```
{
        pole = new double[5];
        com.Send<double[]>(pole, 1, 3);
}
if (com.Randk == 1)
{
        com.Receive<double[]>(0, 3, out pole);
}
```

Výpis 2: Příklad komunikace mezi dvěma procesy [17]

2.4.7 Komunikace mezi více procesy

Jedním z nejčastějších způsobů komunikace je rozesílaní informace z jednoho procesu všem ostatním procesům. Metoda sloužící k této komunikaci se nazývá *Broadcast*. V dalším výpisu (3) je uvedený příklad, kdy proces číslo 3 rozešle pole *double* všem procesům.

```
double[] pole;
  Intracommunicator com = Communicator.world;

  if (com.Randk == 3)
  {
        pole = new double[5];
  }
  com.Broadcast<double[]>(ref pole, 3);
```

Výpis 3: Příklad komunikace typu Broadcast [17]

Velice podobná metoda je *Scatter*. Tato metoda stejně jako *Broadcast* rozesílá data všem procesům, s tím rozdílem, že každý uzel dostává svá vlastní data. Posílá se pole o délce, které odpovídá počtu procesů. Každý proces dostává hodnoty z indexu pole, odpovídající jeho číslu procesu. Další parametr této metody odpovídá číslu procesu, který data odesílá. Vice lze vidět na výpisu 4, kde uzel číslo 2 rozesílá data ostatním uzlům.

```
  Intracommunicator com = Communicator.world;
double[] pole = new double [com.Size];
  double vysledek;

  if (com.Randk == 2)
  {
        for (int i = 0; i < com.Size; i++)
        {
              pole[i] = 2 * i;
        }
  }
  vysledek = com.Scatter<double[]>(pole, 2);
  Console.WriteLine("Muj vysledek: " + vysledek + ", a me cislo procesu: " + Communicator.
      world.Rank);

  // Vypis z konzole
  Muj vysledek: 2, a me cislo procesu: 1
  Muj vysledek: 0, a me cislo procesu: 0.
```

```
Muj vysledek: 4, a me cislo procesu: 2
```

Výpis 4: Příklad komunikace typu Scatter [17]

Opakem k metodě *Scatter* je metoda *Gather*. Tato metoda neodesílá zprávu z jednoho procesu na všechny procesy, ale sbírá data od všech procesů a odesílá je jednomu uvedenému procesu v parametru metody. Na výpisu 5 si lze ve výstupu konzole všimnout různého výstupu jednotlivých procesů.

```
  Intracommunicator com = Communicator.world;
double[] pole = new double [com.Size];
  double cislo = com.Rank * 3;


  com.Gather<double>(cislo, 0, ref pole);


  // com.Rank = 0, obsah pole
  0, 3, 6, 9


  // com.Rank = 1, obsah pole
  0, 0, 0, 0
```

Výpis 5: Příklad komunikace typu Gather [17]

2.4.8 Synchronizace mezi procesy

Může také nastat situace, kdy je nutná synchronizace práce procesu. Ve třídě *Intracommunicator* k tomuto účelu slouží metoda *Barrier*. Tato metoda nemá žádné argumenty, a pokud se zavolá, je důležité, aby ji zavolaly všechny procesy dané aplikace. Pokud proces narazí na tuto metodu, stojí na ní do té doby, dokud ostatní procesy také nedojdou k nejbližšímu volaní metody *Barrier*.

3 Dělení grafu

Na konci minulé kapitoly je vysvětleno, proč je výhodné dělit operace mezi více procesy, a také, že na základě uvedených problémů je důležité nalezení algoritmu, který by rozděloval graf nejenom na základě rovnoměrného rozložení, ale také který se snaží co nejvíce minimalizovat hrany mezi jednotlivými podgrafy. Tímto se sníží množství komunikace. Existuje řada různých algoritmů, které se tímto zabývají. Diplomová práce se věnuje několika z nich.

Nejdříve si projděme *půlení neohodnocených grafu*. Je dán graf množinou vrcholů a hran. Snažíme se najít dvě množiny o stejném počtu vrcholů, jejichž počet hran mezi podmnožinami se blíží nule. Formálnější popis by vypadal takto:

Definice 3.1 *Máme graf G = (V, H), kde |V| je množství vrcholů. Snažíme se najít (| V_1 |, | V_2 |) pocházející z V (t.j., $V_1 \cup V_2 = V$ a $V_1 \cap V_2 = \oslash$), a zároveň*

$$| V_1 | = | V_2 |$$

tak, že

$$|\{e_{ij}; v_i \in V_1, v_j \in V_2\}|$$

je minimální mezi všemi možnými rovnoměrnými rozděleními vrcholů V.

Nyní se podívejme na složitější půlení, a to *půlení ohodnocených grafu*. Snažíme se nalézt dvě množiny se stejným počtem vrcholů, přičemž součet ohodnocení hran mezi dvěma množinami má být minimální.

Definice 3.2 *Máme graf G = (V, H) s ohodnocením hran W, p je počet podmnožin. Snažíme se nalézt p rozdělení (| V_1 |, | V_2 |,...., | V_p |) pocházející z V (t.j.,*

$$\bigcup_{i=1}^{p} V_i = V$$

a $V_i \cap V_j = \oslash$ pro všechny), a zároveň | V_1 | = | V_2 | =....= | V_p | tak, že

$$\sum_{h_i j \in H, v_i \in V_p, v_i \in V_q, p \neq q} W e_{ij}$$

je minimální mezi všemi možnými rovnoměrnými rozděleními vrcholů V.

Nalezení nejideálnějšího rozdělení není vůbec jednoduché, a proto existuje velká řada algoritmů, které se tímto zabývají. Nelze říci, že nějaký algoritmus řeší všechny problémy nejlépe. Je to specifické podle požadavků, kladených na kvalitu a čas. Je nutné si uvědomit, že toto jsou často dva protichůdné parametry. Většinou totiž platí, že čím více je výsledek optimální, tím větší je časová závislost, a naopak, čím rychlejší dělení je, tím je výsledek horší.

Existuje mnoho algoritmů, které jsou časově různě náročné, a poskytuji kvalitativně odlišné výsledky. Tyto algoritmy také přistupují ke své práci odlišně. Některé z nich pracující

s vlastnostmi plotu, některé pouze potřebují vytvořený graf a žádné další informace. Některým stačí lokální pohled a snaha o zlepšení daného rozdělení, některé řeší daný problém globálně. Jsou také algoritmy striktně deterministické, a existují také algoritmy vracející pokaždé tak trochu náhodné výsledky. Najdou se však také takové algoritmy, které řeší problém pomocí grafové teorie. Některé zase přistupují k problému, jako ke speciálnímu problému nějakého jiného problému.

Dále si projděme ukázku různých algoritmů, které jsou vybrány pro to, aby přiblížily danou problematiku. Další algoritmy lze nalézt zde: [11], [6], [7].

3.1 Rekurzivní dělení

Prvním a nejintuitivnějším algoritmem je rekurzivní dělení. Jedná se o půlení grafu na dvě části. Velmi snadno lze pomocí rekurze rozdělit graf na více částí. To se může zdát dosti omezující, ale v podstatě v tomto až takový problém není, protože většina paralelních počítačů má 2^k procesorů. Kdybychom dosáhli ideálního rozpůlení na dvě části, bohužel bychom nedostali tak ideální výsledky, které je možno získat při dělení nějakým *p-way* děličem. Počet useknutých hran mezi částmi by byl vyšší. Na obrázcích níže lze vidět rozdělení rekurzivním dělením a optimalizované rozdělení. Oba případy jsou rozděleny na 8 částí. Rekurzivní dělení Obr. 3a výsledku dosáhlo s rozseknutím 128 hran, a optimalizované rozdělení Obr. 3b pouze 116. Toto samozřejmě znamená nadbytečnou komunikaci. Pro důležité třídy grafů funguje tento algoritmus vcelku dobře, a byť je časová složitost spíše exponenciální, existují také algoritmy používající rekurzivní dělení se složitostí polynomiální. Pokud nám nevadí větší počet rozseknutých hran, můžeme jej relativně dobře využít.

3.2 Rostoucí grafy a hladový algoritmus

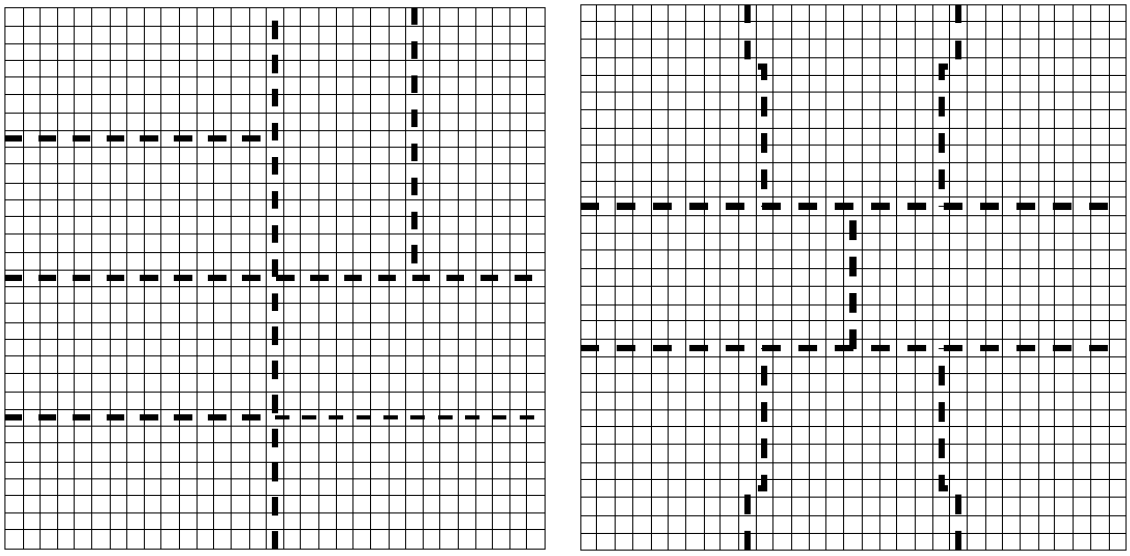
Další z technik, která se nabízí, je tato: Vybrat jeden z vrcholů a s využitím nějaké logiky k němu přidávat další vrcholy. A to až do té doby, dokud rozdělení není dostatečně velké. Tyto algoritmy se nazývají *rostoucí grafy* nebo *hladový algoritmus*. Od sebe se tyto dva způsoby liší již zmíněnou logikou přidávaní dalších vrcholů do podmnožiny. Hladový algoritmus vybírá další vrchol s ohledem na to, aby počet řezů hran rostl co nejméně. Rostoucí grafy přidávají další vrcholy tak, aby výsledný podgraf určitým způsobem rostl. Některé jiné rostoucí podgrafy začínají s více začínajícími vrcholy, a ty paralelně rostou. Některé rozdělení mohou mít i větší prioritu. Pro ukázku je zde uveden jeden z pseudokódu hladového algoritmu 1. Výběr počátečního vrcholu je uveden zde: [9].

Tyto algoritmy se vyznačují svou rychlostí. Oproti rekurzivnímu půlení je jejich výhodou to, že graf dělíme na pod-části přímo. Z toho důvodu je čas potřebný pro dělení v podstatě nezávislý na počtu podgrafů. Nevýhodou je množství oddělených hran, těch je často zbytečně mnoho, a to především u složitých grafů. Toto dělení je také citlivé na výběr počátečního vrcholu. Díky rychlosti je však možné si dovolit spustit algoritmus s rozdílnými počátečními body vícekrát, a poté vybrat to nejlepší rozdělení.

Obrázek 3: Rozdělení grafu na 8 částí, dvěma různými způsoby [3]

(a) Rekurzivní dělení

(b) Optimalizované dělení



3.2.1 Kernighan-Lin algoritmus

Jedná se o jeden z nejrychlejších algoritmů využívající se k optimalizaci existujících rozdělení. Původně bylo vybráno čistě náhodné rozdělení, na které byl Kernighan-Lin (K/L) aplikován. Toto se několikrát zopakovalo, a nakonec se vybral nejlepší výsledek. Pro velké grafy to však není příliš efektivní. Nyní se pro vylepšení používá rozdělení, které je nalezené nějakým jiným algoritmem. K/L se dívá na problém "lokálně", snaží se snížit počet rozdělených hran tím, že vyměňuje mezi rozděleními jejich sousední uzly. Tento algoritmus tak doplňuje problémy dělicích algoritmů, které se dívají na problém globálně, a ignorují lokální problémy.

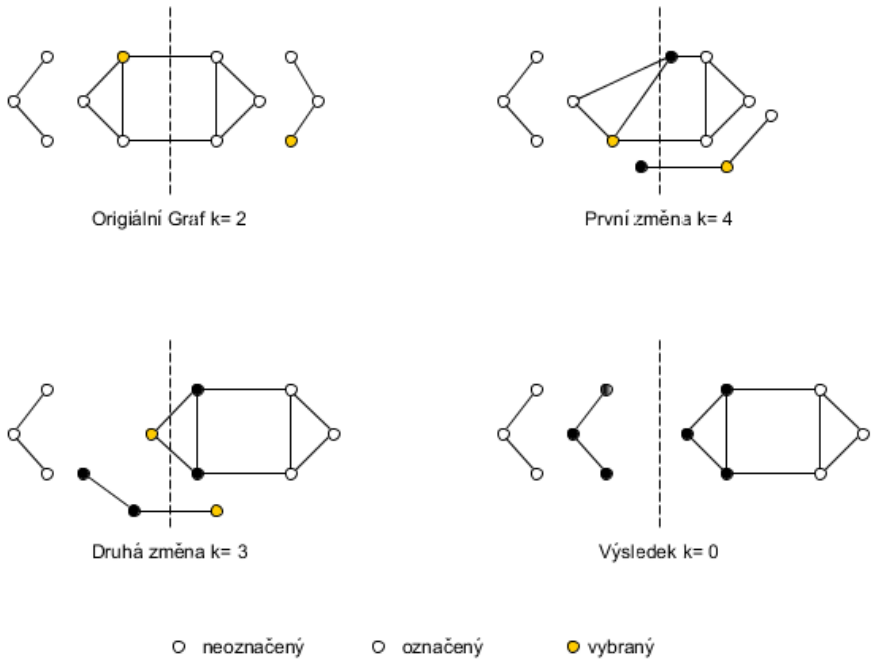
Algoritmus spočívá v tom, že v rámci jednoho kroku najde v jedné množině vrcholů takový vrchol, jehož přemístěním se počet rozdělených hran co nejlépe změní. Tento vrchol lze nalézt i ve druhé množině. Tyto vrcholy poté navzájem vyměníme. Po jedné výměně se může počet rozdělených hran zvýšit. Algoritmus se však kvůli tomu nezastaví, počítá s tím, že se v dalších krocích najdou lepší ohodnocení. Vrcholy se vyměňují tak dlouho, dokud se nejmenší množina vrcholů zcela nepřemístí do jiné množiny.

Na obrázku níže, Obr. 4 vidíme příklad práce KL algoritmu. Na počátku máme dvě rozdělení, které jsou propojené dvěma rozdělenými hranami $k_0 = 2$. Z každé množiny byl vybrán jeden vrchol, jehož přesunutím se získalo, ze všech možných přesunutí, nejlepší ohodnocení. Stav po prvním přesunutí lze vidět vpravo nahoře. Je patrné, že to zatím moc nepomohlo. Nyní máme 4 rozdělené hrany. Algoritmus však pokračuje dále, zbývají ještě 2 vrcholy z původního nejmenšího rozdělení. Přesunutí se provádí ještě dvakrát, na obrázku vpravo dole lze vidět výsledné rozdělení s 0 rozdělenými hranami.

Vstup: graf G = (V, H);
Výstup: grafy $G_1 = (V_1, H_1), G_2 = (V_2, H_2)$;
Odznač všechny vrcholy;
Vyber nějaký pseudo-okrajový vrchol (př: jeden z vrchol ze dvou, které jsou od sebe v grafu vzdálené čím jak nejdále), jako začínající vrchol označte ho a přidejte k prvnímu podgrafu;
for *ještě nebyly naplněny všechny podgrafy* **do**
    while *aktuální podgraf není plný* **do**
        Vyberte jeden vrchol podle nejlépe ohodnocené hrany ze sousedních neoznačených vrcholu;
        Označte ho a přidejte k aktuálnímu podgrafu;
    end
    Pokud ještě existují neoznačené vrcholy, vyberte jeden nejlépe ohodnocený sousedící vrchol k předešlému rozdělení a označte ho jako počáteční nové rozdělení;
end

Algoritmus 1: Ukázka hladového algoritmu v pseudokódu

Obrázek 4: Kernighan-Lin příklad [3]



Algoritmus popsaný výše (a implementovaný zde: [10]) má složitost $O(\mid m \mid)$ jednu

iteraci, kde m je počet hran v grafu . Existuje celá řada vylepšení, některé z nich jsou popsány zde:([12], [13], [11]).

3.3 Spektrální dělení

Spektrální dělení je odlišné od těch, na které jsme se doposud dívali. Nepracuje s grafem jako takovým, ale s jeho matematicky upravenou reprezentací. Spektrální dělení je specifické v tom, že narozdíl od ostatních dělení, využívá plovoucí desetinou čárku. Vzhledem k tomu, že každý vrchol vychází z celého grafu, lze říci, že toto dělení má globální pohled na věc. Následující teorie o *spektrálním dělení* vychází z této diplomové práce: [16].

Definice 3.3 *Nechť $G = (V, H)$ je graf s $n =\mid V \mid$ vrcholy očíslovanými nějakým způsobem, a s $deg(v_i)$ to je stupeň vrcholu i (t.j. počet přilehlých vrcholů). Laplaceova matice $L(G)$ je $n \times n$ symetrická matice s jedním řádkem a sloupcem pro každý vrchol.*

- *Obsah je definován takto:*

  l_{ij} = *-1 : pokud* $i \neq j$ *a* $h_{ij} \in H$

  l_{ij} = *0 : pokud* $i \neq j$ *a* $h_{ij} \notin H$

  $l_{ij} = deg(v_i)$ *: pokud* $i = j$

Vzhledem k tomu, že *Laplaceova matice* je symetrická, jsou všechny její vlastní hodnoty reálné. Z Gershgorin kruhového teorému vyplývá, že všechna vlastní čísla jsou nezáporná. L(G) je tedy pozitivně semidefinitní.

Matice L(G) a A(G) jsou ve vztahu

$$\mathrm{L} = D - A,$$

D je diagonální matice, kde deg_{ii} je stupeň vrcholu i.

Vlastní hodnoty matice L(G) jsou seřazeny $\lambda_0 = \lambda_1 \leq ... \leq \lambda_{n-1}$. Vlastní vektor odpovídající λ_0 je vektor stejných hodnot. Vícenásobnost λ_0 je rovna číslu souvislých komponent grafu. Druhá nejmenší vlastní hodnota λ_1 je větší než nula, pokud G je souvislý graf. Základní vlastnosti, spektrum a vztah k mnoha grafovým invariantům Laplaceovy matice se zvláštním důrazem na λ_1 lze dále najít zde: [14], [15]. Fiedler toto nazývá *algebraic connectivity*. Fiedler také zkoumal vlastnosti teorie grafu související s vlastním vektorem, vycházející z λ_1, neboli druhý vlastní vektor (tomuto vektoru se říká Fidlerův vektor). Souřadnice tohoto vektoru jsou přirozeným způsobem přiřazeny vrcholům z grafu G. Což může být považováno jako ohodnocení vrcholů. Toto ohodnocení se nazývá *charakteristické ohodnocení grafu G*.

3.3.1 Spektrální půlení

Tato podčást je věnována způsobu, jak pomocí spektrálního dělení rozpůlit graf na dvě části. Snažíme se o to, aby tyto části měly přibližně stejný počet vrcholů, a to za předpokladu, že počet rozdělených hran je minimální.

Mějme tedy graf *G=(V, H)* a Fiedlerův vektor $\vec{v} = (v_0, v_1, ..., v_n)$. Dále předpokládáme, že graf G je souvislý, a platí tedy pro něj výše uvedené vlastnosti. Hlavní myšlenkou je nalézt *rozdělující hodnotu s*. S ohledem na Fiedlerův vektor můžeme rozdělit vrcholy na dvě podmnožiny (V_1, V_2). První množina V_1 odpovídá $v_i \leq s$, ta druhá V_2 splňuje tuto podmínku $v_i > s$. Říká se tomu *Fiedlerův řez*. Za s se dosazuje medián. Je nutno snažit se rozdělit množiny tak, aby se v nich počet vrcholů lišil maximálně o jeden. Ještě před ukázkou algoritmu 2 je potřeba vysvětlit pár značek.

Máme dvě množiny vrcholů $V_1^{'}$ a $V_2{'}$. $V_1^{'}$ jsou vrcholy z V_1 s tím, že minimálně jedna hrana z každého vrcholu je propojena s vrcholy z množiny V_2. A opačně, vrcholy z V_2, které jsou propojeny s V_1 značíme $V_2{'}$. Samozřejmě existuje množina hran $H^{'}$, která propojuje vrcholy $V_1^{'}$ a $V_2{'}$. Tato množina hran je vlastně množina rozdělujících hran, rozdělující graf G na dva podgrafy.

Vstup: graf G = (V, H);
Výstup: grafy $G_1 = (V_1, H_1), G_2 = (V_2, H_2)$;

1. vypočítat Fiedlerův vektor;
2. najít medián z vektoru v;
3. pro každý v_i z V
   (a) if $v_i \leq$ medián;
   přidej vrchol do množiny V_1;
   (b) jinak;
   přidej vrchol do množiny V_2;
4. if $\mid V_1 \mid - \mid V_2 \mid > 1$;
   přesuň vrcholy rovny mediánu z V_1 do V_2, za účelem udělat rozdíl maximálně rovný 1;
5. nechť $V_1^{'}$ je množina vrcholů ve V_1, propojená s vrcholy V_2 ;
   nechť $V_2^{'}$ je množina vrcholů ve V_2, propojená s vrcholy V_1 ;
   vytvořit množinu $H^{'}$ - množinu hran z G s jedním bodem ve $V_1^{'}$ a druhým ve $V_2^{'}$;
6. nechť E_1 je množina hran s oběma vrcholy ve V_1;
   nechť E_2 je množina hran s oběma vrcholy ve V_2;
   vytvořit grafy $G_1 = (V_1, H_1), G_2 = (V_2, H_2)$;
7. konec.

Algoritmus 2: Ukázka spektrálního dělení

3.3.2 Dělení na *k* podgrafů

Mějme graf $G = (V, H)$, $n =\mid V \mid$, kde množinu vrcholů V dělíme na části tak, že v každé vytvořené množině bude přibližně n/k vrcholů. Hlavní myšlenka je taková, že spektrální dělení budeme aplikovat tak dlouho, dokud nebudeme mít rovnoměrné rozdělení o k částech. Jsou tedy dva různé případy. Pokud $k =$ mocnina 2, pak lze použít rekurzivní půlení. Ve druhém případě je nutno použít modifikovanou podobu rekurzivního půlení. První z možných případů 3 je nyní uveden.

Vstup: graf G = (V, H); k = číslo označující na kolik podgrafů máme graf rozdělit ;
Výstup: grafy $G_1 = (V_1, H_1), ...G_k = (V_k, H_k)$;

1. aplikuj spektrální dělení 2 na graf(G) pro nalezení (G_1, G_2);
2. if $(k/2 > 1)$
   (a) Rekurzivní dělení $(G_1, k/2)$
   (b) Rekurzivní dělení $(G_2, k/2)$
3. return $G_1, ...G_k$;
4. konec.

Algoritmus 3: Spektrální rekurzivní půlení

Vzpomeňme si, že pro spektrální dělení na dvě části byl použit medián. Pro dělení, kde se k nerovná mocnině 2, je nutno použít jiný kvantil. Ukázky takto modifikovaných algoritmu jsou zde [16].

V textu výše, je již popsán základní přehled o spektrálním dělení. Tímto dělením se však nadále tato práce nezabývá. V případě zájmu o tuto problematiku, lze na tomto odkazu [16] nalézt více informací o tomto dělení, a jeho dalších modifikacích.

3.4 Random walk

Náhodná procházka je dělení, kterému se tato práce velmi věnuje a pokouší se jej zefektivnit. V grafu představuje přirozený stochastický proces, založený na Markovém řetězci 3.4.1. Pro popis lze velmi dobře využít *Návrat dezorientovaného opilce z restaurace domů*. Opilec se na každé křižovatce rozhoduje, jakou cestou se vydá, a odhaduje se s jakou pravděpodobnosti se opilec vrátí do rána domů. Následující popis RW a jeho realizace je čerpána z těchto zdrojů: [26], [27], [28].

3.4.1 Markovovy řetězec

Markovův řetězec se zabývá přechody mezi stavy a vyčísluje pravděpodobnosti těchto přechodů. Máme množinu stavů: $S = s_1, s_2, s_3, ..., s_n$, začíná se v jednom z těchto stavů. Z tohoto stavu chceme přejít do jiného, tomuto přechodu říkáme *krok*. Každý *krok*, který

začíná ve stavu s_i a končí ve stavu s_j s pravděpodobností p_{ij} je nezávislý na předešlém kroku. Pravděpodobnosti p_i říkáme pravděpodobnost přechodu, nebo přechodová pravděpodobnost. Počáteční distribuce pravděpodobnosti specifikuje pouze počáteční stav, to znamená, že počáteční distribuce pravděpodobnosti je vektor, jehož prvek i reprezentuje počáteční stav a je roven jedné, ostatní jsou rovny nule.

3.4.2 Přechodová matice

Pravděpodobnosti přechodů se obvykle zaznamenávají v *přechodové matici*. V této matici každý řádek, či sloupec odpovídá jednomu konkrétnímu stavu z množiny S. Velikost této matice je | S | x | S |. Prvek p_{ij} je pravděpodobnost přechodu ze stavu s_i do stavu s_j. Také platí pravidlo, že součet hodnot v každém sloupci je roven jedné.

Definice 3.4 *Nechť P^n je pravděpodobnost taková, že pro ij-tý prvek matice podle Markovového řetězce se po n krocích dostane z počátečního stavu s_i do koncového stavu s_j.*

Pro lepší vysvětlení je na příkladu 3.1 ukázaná definice 3.4.

Příklad 3.1
Máme množinu stavu $S = s_1, s_2, s_3$. Přechodová matice P vypadá následovně:

$$P = \begin{pmatrix} p_{11} & p_{12} & p_{13} \\ p_{21} & p_{22} & p_{23} \\ p_{31} & p_{32} & p_{33} \end{pmatrix}$$

Hodnoty pravděpodobnosti p_{ij} mohou být v intervalu $< 0, 1 >$. Nula znamená, že mezi stavy s_i a s_j není provázaní takové, abychom se jedním krokem dostali z jednoho stavu do stavu druhého. Pokud zjišťujeme pravděpodobnost přechodu ve druhém kroku, pak vynásobíme řádek počátečního stavu se sloupcem koncového stavu (což je skalární součin). Opakováním tohoto procesu n-krát získáme pravděpodobnost přechodu ze stavu s_i do stavu s_j n-tým krokem, značíme jako p_{ij}^n ■

S tímto se pojí několik důležitých pojmů. Pokud máme vypočítat pravděpodobnost po n-krocích, tak i-tý řádek je nazýván *distribucí pravděpodobností* markovova řetězce. Začíná v počátečním stavu s_i a do všech ostatních stavů se dostane n-tým krokem. Komponenta j tohoto řádkového vektoru p_i nám tedy říká pravděpodobnost, že se po n krocích dostaneme do konečného stavu s_j.

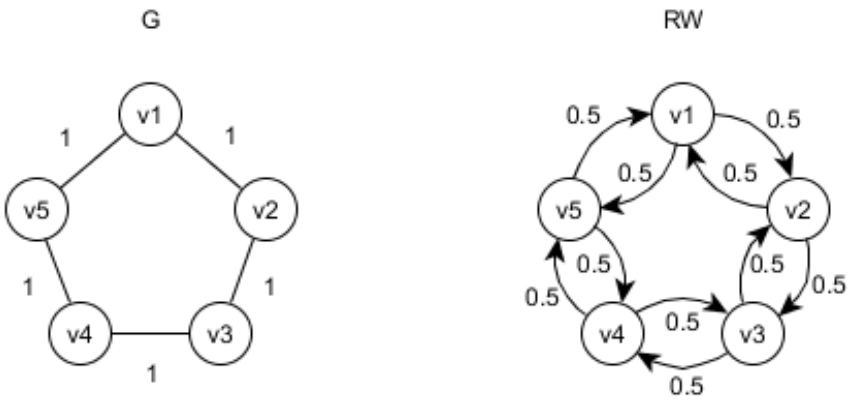
Další dva pojmy jsou *stacionární distribuce* a *doba míchání*. Tyto pojmy se týkají distribuce pravděpodobnosti, která se dostane do neměnitelného stavu. To znamená, že v dalších krocích se již prvky vektoru nemění. Tento stav se nazývá *stacionární distribuce*. Doba, díky které jsme se dostali do tohoto stavu, se jmenuje *doba míchání*.

3.4.3 Random Walk v grafech

Abychom mohli použít metodu *random Walk*, převedli jsme zadaný graf na RW. A to proto, abychom získali vhodnější ohodnocení grafu. Výsledný graf RW bude mít stejné vrcholy,

jen ohodnocení bude odlišné. Vrcholy si tedy představujeme jako stavy a hrany jako přechody. Všechny hrany ohodnotíme pravděpodobností přechodu mezi vrcholy, tedy hranu h_{ij} ohodnotíme pravděpodobností přechodu mezi vrcholy v_i a v_j. Tak získáme normalizaci v ohodnocení hran. Každé ohodnocení bude v intervalu $< 0, 1 >$, a součet inidentních s libovolným vrcholem je vždy roven jedné. Příklad takovéhoto převodu lze vidět na obrázku 5.

Obrázek 5: Příklad převodu grafu G na RW graf



Každá neorientovaná hrana se nám díky komutativnosti rozdělí na dvě orientované hrany. A to z důvodu, že pravděpodobnost hrany z v_i do v_j nemusí být stejná jako z v_j do v_i. V našem případě však stejná je.

Na základě RW grafu jsme tedy vytvořili matici přechodu tak, jak jsme již naznačili 3.4.2. Matici můžeme vyčíslit několika způsoby. Jeden způsob je podle ohodnocení hran, kdy hodnota prvků matice p_{ij} je definována jako poměr hodnoty hran mezi vrcholy i a j, oproti součtu hodnot všech hran incidentních s vrcholem i.,

$$p_{ij} = \frac{w(i,j)}{\sum_{k=0}^{n} w = (i,k)}$$

, kde n je počet vrcholů grafu G.

3.4.4 Nové ohodnocení algoritmem Neighbour Similarity

Tento algoritmus se zaměřuje na porovnávání podobností mezi dvěma sousedními vrcholy. Dále v definici 3.5 zavádíme *pravděpodobnost návštěvnosti*. Tento algoritmus byl převzat z toho zdroje: [26].

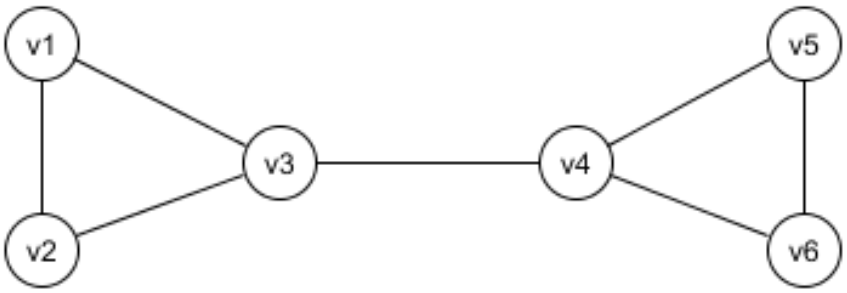
Definice 3.5 *$P^k visit(v) \in R^n$je i-tý řádek přechodové matice P^k, kde k je k-tá mocnina matice P. $P^k visit(v)$ obsahuje informaci, že z vrcholu v_i se dostaneme do jiného vrcholu v k-tém kroku s pravděpodobností p(j), kde p(j) je položka z $P^k visit(v)$.*

Tento vektor pomáhá popsat souvislost mezi samotným vrcholem a jeho okolím, které je závislé na struktuře grafu. Je logické, že $P^{\infty}visit(v)$ je rovno *statické distribuci* v začátku ve vrcholu v. V našem případě je využíváno toto:

$$P^{\leq k}visit(v) = \sum_{i=1}^{k} P^{i}visit(v).$$

Příklad 3.2
Máme zadaný následující graf:

Obrázek 6: Příklad grafu převáděný do NS



$$A = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 \end{pmatrix}$$

$$P = \begin{pmatrix} 0 & 0.5 & 0.5 & 0 & 0 & 0 \\ 0.5 & 0 & 0.5 & 0 & 0 & 0 \\ 0.33 & 0.33 & 0 & 0.33 & 0 & 0 \\ 0 & 0 & 0.33 & 0 & 0.33 & 0.33 \\ 0 & 0 & 0 & 0.5 & 0 & 0.5 \\ 0 & 0 & 0 & 0.5 & 0.5 & 0 \end{pmatrix}$$

Na grafu lze vidět, že se skládá ze dvou podgrafů propojených vrcholy v_3 a v_4. Vyzkoušíme nejdříve podobnost dvou vrcholů ve stejném podgrafu, a to v_1 a v_2, za k dosadíme 2.

$$P^{\leq 2}visit(v1) = (1, 1.13, 1.36, 0.42, 0.06, 0.06)$$
$$P^{\leq 2}visit(v2) = (1.13, 1, 1.36, 0.42, 0.06, 0.06)$$

Jen při pohledu na tyto dva vektory je možno vidět, že si jsou velice podobné. V případě, že použijeme porovnávání založené na vzdálenosti vektoru, vyjde nám podobnost

0.786, při cosínové podobnosti 0.99869. Tento výsledek je velmi dobrý, a není tedy pochyb o tom, že tyto dva vektory patří do jednoho podgrafu. Jako na další pár v tomto příkladu se podívejme na vrcholy v_3 a v_4, opět pro $k = 2$.

$$P^{\leq 2}visit(v3) = (0.9, 0.9, 1.06, 0.59, 0.28, 0.28)$$

$$P^{\leq 2}visit(v4) = (0.28, 0.28, 0.59, 1.06, 0.9, 0.9)$$

Tyto vektory jsou přesně zrcadlové díky symetrickému rozloženi grafu. I přesto však lze vidět, že jsou grafy hodně odlišné. Výpočtem se tato odlišnost jen potvrdila. Podle výsledku na základě vzdálenosti a cosinusu jsme dostali tyto výsledky 0,078 a 0.6953. Na základě těchto údajů lze konstatovat, že tyto vektory nepatří do stejného shluku. ■

Definice 3.6 *Nechť G(V, H, w_s) je ohodnocený graf a k je malá konstanta. Přehodnocení grafu G podle Neighbour Similarity (NS(G)) je definováno takto:*

$$NS(G) = G_s(V, H, w_s),$$

$$kde, \forall h(u, v) \in H, w_s(u, v) = sim(P^{\leq k}visit(u), P^{\leq k}visit(v))$$

$sim(P^{\leq k}visit(u), P^{\leq k}visit(v))$ znamená míru podobnosti mezi vektory $P^{\leq k}visit(u)$ a $P^{\leq k}visit(v)$, jejichž hodnota se zvětšuje, pokud si jsou více podobné.

Na závěr si ukažme pseudo-algoritmus pohromadě. Více informací, lze nalézt zde [26].

Vstup: graf G = (V, H); k = nějaké malé přirozené číslo ;
Výstup: přehodnocený graf G= (V, H) ;

1. Pro všechny hrany $h \in H$:
   (a) Spočítej vektory $P^{\leq k}visit(v)$ a $P^{\leq k}visit(u)$ (kde u,v jsou incidentní s $h \in H$)
   (b) Vypočti podobnost mezi vektory $P^{\leq k}visit(v)$ a $P^{\leq k}visit(u)$ a ulož ji do pomocné struktury pod klíčem definující hranu $h \in H$
2. Pro všechny hrany $h \in H$:
   (a) Z pomocné struktury vyber hodnotu vypočtené podobnosti pro hranu $h \in H$ a nahraď ohodnocení této hrany touto hodnotou
3. konec.

Algoritmus 4: Algoritmus neighbour similarity

3.4.5 Random walk pro implementaci bez MPI

Po několika neúspěšných hledáních *random walk* algoritmu s možností využití knihoven MPI, jsme se rozhodli, že zkusíme naimplementovat vlastní intuitivní algoritmus, vycházející z myšlenky *random walku*. V následující části je uveden návrh tohoto algoritmu.

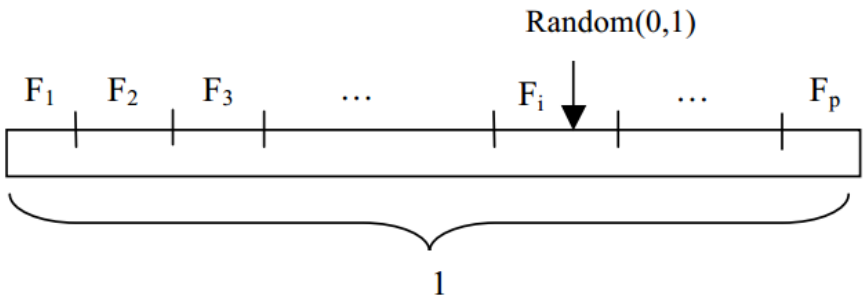
3.4.5.1 Nalezení RW okolí První část tohoto algoritmu řeší nalezení samotných RW okolí 3.7. Na vyřešení překryvů se soustředí druhá část pro řešení konfliktů.

Definice 3.7 *RW okolí je skupina vrcholů, které se nacházejí do určité vzdálenosti v od jednoho centrálního vrcholu. Tato skupina je výsledkem algoritmu random walk. Jednotlivé skupiny mají podobné vlastnosti jako shluky, ale mohou se navzájem překrývat.*

Výběr vrcholů z množiny vrcholů, které doposud nejsou přiřazeny k žádnému RW okolí, je náhodný. V případě, že je tato množina prázdná, přechází se k řešení konfliktů. Tento vrchol je označen jako počáteční a bude z něho spuštěno s sledů o k krocích. To znamená, že vezmeme počáteční vrchol a váhy hran vedoucích z něj do jiných vrcholů, a pomoci rulety (viz. [19]) se určí do jakého dalšího vrcholu se vydáme. Z nově navštíveného vrcholu hledáme další vrchol stejným způsobem. Toto procházení končí, pokud v řadě navštívíme k vrcholů. Tyto vrcholy nemusí být jedinečné. Může se stát, že se do již navštíveného vrcholu vrátíme, a to buď přímo, nebo nepřímo přes nějaký třetí uzel. Vytvořený sled uložíme a vytvoříme stejným způsobem dalších s-1 sledů.

Náhodný výběr pomocí rulety (viz obrázek 7) v našem případě funguje takovým způsobem, že vygenerujeme číslo od 0 do 1 a přiřadíme toto číslo odpovídající hraně. Podíly vah jsou totiž za sebou rozprostřeny v intervalu od 0 do 1. Pokud máme tedy váhy hran $h_1 = 0.1$, $h_2 = 0.3$ a $h_3 = 0.6$ a náhodně vygenerujeme číslo 0.2, pak vybereme hranu h_2, protože se nachází v intervalu od 0.1 do 0.4.

Obrázek 7: Výběr hrany pomocí rulety



Nyní se nabízí dvě možnosti jak vybrat vrcholy z vytvořených sledů. První, ta jednodušší, je vybrat do nového RW okolí ty vrcholy, které se ve sledech vyskytly vícekrát než jednou. Druhá možnost je založena na $P^k visit(v)$, kde k z $P^k visit(v)$ je rovno našemu k (počet kroků). Nebudeme však počítat k matic $P^k visit(v)$, ale budeme započítávat jen ty pravděpodobnosti, které při RW okolí využijeme. Pokud tedy využijeme graf z příkladu Neighbour Similarity 6, a vygenerují-li se tyto tři níže uvedené sledy, tak bude zřejmé, že počáteční vrchol je v_1. Přiřadíme mu proto hodnotu návštěvnosti rovnou 1 a první sled zahájíme přechodem s pravděpodobností 0.5 do vrcholu v_2. K v_2 můžeme tedy přiřadit hodnotu návštěvnosti 0.5. Ve druhém kroku jsme se přesunuli do vrcholu v_3 s pravděpodobností 0.5. K vrcholu v_3 jsme však přiřadili hodnotu 0.25, což je součin pravděpodobnosti

přechodu od počátečního vrcholu k aktuálnímu. V posledním kroku jsme se vrátili do počátečního vrcholu s hodnotou návštěvnosti 0.0825. Po ukončení dalších dvou sledů sečteme pro každý vrchol hodnoty návštěvnosti, poté vypočítáme průměr těchto výsledných hodnot. Vrcholy, jejichž součet hodnot návštěvnosti je nadprůměrný, jsou vybrány do výsledného RW okolí. Do výpočtu průměru není započítána hodnota počátečního vrcholu, protože je do výsledného RW okolí zahrnut automaticky.

$v_1 \overset{0.5}{\rightarrow} v_2^{0.5} \overset{0.5}{\rightarrow} v_3^{0.25} \overset{0.33}{\rightarrow} v_1^{0.0825}$
$v_1 \overset{0.5}{\rightarrow} v_3^{0.5} \overset{0.5}{\rightarrow} v_1^{0.25} \overset{0.5}{\rightarrow} v_3^{0.0125}$
$v_1 \overset{0.5}{\rightarrow} v_3^{0.5} \overset{0.33}{\rightarrow} v_4^{0.165} \overset{0.33}{\rightarrow} v_5^{0.054}$

Následující pseudokód popisuje algoritmus 5 pro nalezení RW okolí. V tomto algoritmu lze vidět, že počet sledů vytvořených pro nové RW okolí nemusí být vždy stejný, a to v případě, že stupeň počátečního vrcholu je vetší než výchozí počet sledů. V takovém případě se počet sledů nastaví rovno stupni vrcholu. Pokud se i tak stane, že se do RW okolí přidá pouze počáteční vrchol, pak se počet sledů zdvojnásobuje až do té doby, dokud nedostaneme více než jeden vrchol.

```
Vstup: graf G = (V, H);
k= počet sledů pro každý RW;
s= počet kroků v každém RW ;
Výstup: RW okolí S_1 = (V_1, H_1), ....S_n = (V_n, H_n), kde může nastat V_j = V_i,
V_j ∈ S_j a Vi ∈ S_i a S_i ≠ S_j;
while existuje vrchol, který není v žádném RW okolí do
    Vybrat náhodně počáteční vrchol, který zatím nenáleží žádnému RW okolí;
    if stupeň počátečního vrcholu je větší než k then
        počet sledu k bude roven stupni vrcholu;
    end
    while RW okolí obsahuje pouze počáteční vrchol do
        for vytvořit k sledu pro aktuální RW okolí do
            Provést RW o s-krocích na každém počátečním uzlu;
        end
        vybrat z vytvořených sledů vrcholy do RW okolí;
        if RW okolí obsahuje pouze počáteční vrchol then
            zdvojnásobit počet sledů k;
        end
    end
end
```

Algoritmus 5: Vytváření RW okolí

3.4.5.2 Řešení konfliktů Aby bylo možné získat shluky, ve kterých je každý vrchol přiřazen pouze do jednoho shluku, je třeba RW okolí vytvořené pomocí RW zpracovat. Nejdříve dochází ke sjednocení. Tím se vytvoří větší RW okolí. Pokud však i po sjednoco-

vání existují vrcholy, které jsou přiřazeny do více než jednoho RW okolí, pak je odpovídajícím algoritmem určeno v jakém RW okolí zůstane vrchol zachován.

V první části dochází ke snaze sjednotit vytvořené RW okolí, a to na základě počtu společných vrcholů. Následuje příklad, který je založen na zadaném parametru a udává, že při 45 % společných vrcholů bude RW okolí sjednoceno. Máme dvě RW okolí, v prvním je 9 vrcholů a ve druhém 6, víme, že 4 vrcholy se nachází v obou RW okolích. Výsledné podíly tedy jsou 0.444 a 0.666. Na základě prvního podílu z prvního RW okolí bychom nesjednocovali, ale z důvodu, že $0.666 > 0.45$ jsou RW okolí sjednocena.

Druhá část řešení konfliktů záleží na tom, zda byly při vytváření RW okolí používány pro výběr vrcholů výskyty ve sledech, nebo hodnoty návštěvnosti. Nejprve je tedy třeba vysvětlit výskyty ve sledech.

V případě, že nějaký konfliktní vrchol náleží do více RW okolí, jsou jednotlivé RW okolí projity a je zjištěno v kolika sledech se v daném RW okolí vyskytl. Pokud má nějaký RW okolí více výskytů než ostatní, pak mu tento konfliktní vrchol zůstane, a v ostatních je odstraněn. Za předpokladu, že tato metoda nenalezne pouze jeden RW okolí s největším počtem výskytů, je tento vrchol vybrán jako počáteční a je pro něj vytvořeno vlastní pseudo RW okolí. Vytvořené RW okolí však nepřidáme k ostatním, ale použijeme jej pouze k tomu, abychom určili do jakého RW okolí vrchol přidáme. Toto nastane tak, že pro každé RW okolí, který obsahuje náš konfliktní vrchol. Dále je spočítáno v kolika vrcholech se prolíná s pseudo RW okolím. RW okolí, které se prolíná s pseudo RW okolím nejvíce je konfliktní vrchol zachován. Pokud je počet RW okolí s nejvyšším překryvem vyšší než jeden, je vítěz, ve kterém vrchol zachováme, vybrán náhodně. Jednodušší je však způsob, kdy jsou známy hodnoty návštěvnosti. V takovém případě je přiřazen konfliktní vrchol RW okolí, které má největší hodnotu návštěvnosti. Pokud by se stalo, že hodnota bude stejná u více RW okolí, je přiřazen konfliktní vrchol náhodně jednomu z nich. Jakmile každý konfliktní vrchol náleží pouze jedinnému RW okolí, pak lze prohlásit, že množina RW okolí grafu odpovídá množině shluků.

Algoritmus 6 je variantou algoritmu s počtem výskytů vrcholů ve sledech.

3.4.6 Random walk pro implementaci s MPI

Random walk algoritmus této práce využívá MPI knihovny pouze v části pro nalezení RW okolí. Část řešení konfliktu tedy zůstává stejná, protože by ji nebylo možné předělat na více výpočetních uzlů, aniž by komunikace mezi uzly nezpomalila výpočet na tolik, že by byl pomalejší než sekvenční výpočet.

3.4.6.1 Nalezení RW okolí s MPI Tuto část lze opět řešit dvěma způsoby. U prvního z nich je opět očekáváváno, že bude pomalejší z důvodu velké komunikace mezi uzly.Tento způsob řešení spočívá v tom, že pro každý vygenerovaný počáteční uzel je vytváření sledů rovnoměrně rozděleno na jednotlivé uzly. Z těchto sledů se nakonec pokusí řídící uzel složit nové RW okolí.

Druhou možností je, že řídicí uzel vygeneruje n-1 počátečních vrcholů (kde n je počet výpočetních uzlů) a rozešle je mezi ostatní uzly. Ty řídicímu uzlu vrátí již vytvořené RW okolí. Řídicí uzel pak zjistí další vrcholy nepřiřazené žádnému RW okolí a rozešle je. Toto

Vstup: RW okolí $S_1 = (V_1, H_1), ...S_n = (V_n, H_n)$, kde může nastat $V_j = V_i$, $V_j \in S_j$ a $Vi \in S_i$ a $S_i \neq S_j$;
p= minimální procentuální shodnost RW okolí pro sjednocení ;
Výstup: shluky $S_1 = (V_1, H_1), ...S_n = (V_n, H_n)$, kde nemůže nastat $V_j = V_i$, $V_j \in S_j$ a $Vi \in S_i$ a $S_i \neq S_j$;

```
while bylo provedeno sjednoceni do
    for pro každé RW okolí do
        pokusit se RW okolí sjednotit s jiným RW okolím na základě společných
        uzlů (společné uzly > p);
    end
end
for pro každý vrchol do
    if nachází se konfliktní vrchol ve více RW okolí then
        zjistit v kolika sledech se nachází konfliktní vrchol v jednotlivých RW okolí;
        if počet výskytu vrcholů ve sledech RW okolí je stejný then
            vytvořit pseudo RW okolí s počátečním bodem rovným konfliktnímu
            vrcholu;
            zjistit nejčastější se vyskytující RW okolí, obsahující vrcholy z
            vytvořeného pseudo RW okolí;
            if pokud RW okolí s největším obsahem vrcholů z pseudo RW okolí je
            více then
                vyber z těchto RW okolí náhodně vítěze;
            end
        end
        odstraň konfliktní vrchol ze všech RW okolí, kromě vítězného;
    end
end
```

Algoritmus 6: Řešení konfliktů

se bude dít tak dlouho, dokud je počet nepřiřazených uzlů větší, než počet výpočetních uzlů. Poté jsou chybějící počáteční vrcholy vyberány z vrcholů, které jsou již obsaženy v jiném RW okolí. Po přijetí těchto posledních vytvořených RW okolí se přesuneme do řešení konfliktů.

3.5 Parmetis

Jeden z nejpoužívanějších programů současnosti pro dělení rozsáhlých grafů je *Metis*, a jeho obdoba využívající MPI komunikaci *Parmetis*. Tyto programy přistupují k dělení grafu vlastním způsobem. Popisem těchto algoritmů se zabývá následující zdroj [20].

3.5.1 Metis

Princip dělení grafů v Metisu spočívá v tom, že graf nejdříve *zhrubneme*, pak je na něm provedeno *počáteční rozdělení*, a nakonec se z rozděleného hrubého grafu získá původní rozdělený graf. Této části se říká *zjemnění*.

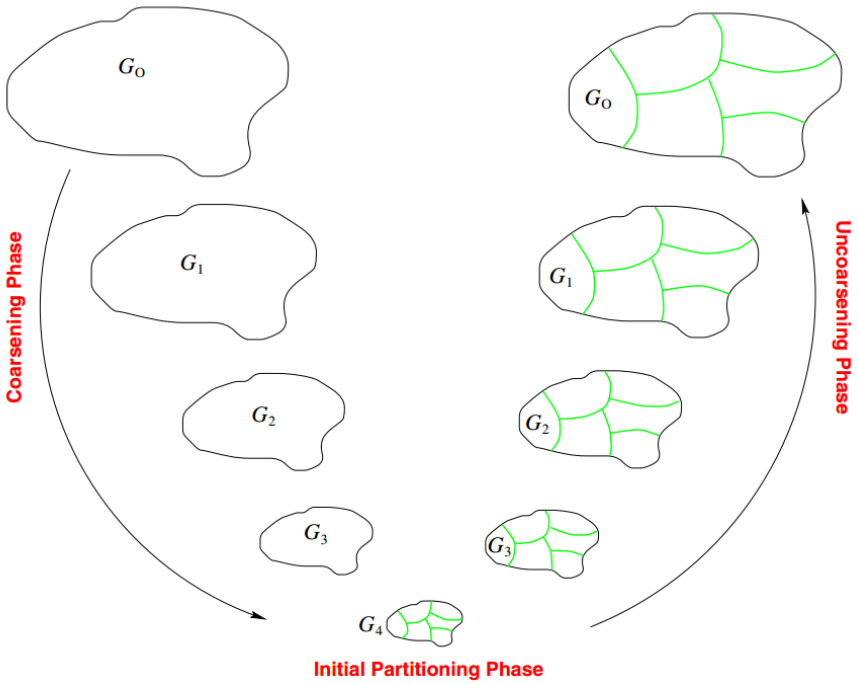
3.5.1.1 Zhrubnění (Coarsening) Princip zhrubnění je ve snaze vytvořit malý graf ze zadaného vstupního grafu. Toho je dosaženo tím, že je sestrojena sekvence grafů, z nichž každý je vytvořen slučováním vrcholů do jednoho z předešlého grafu. Pokud jsou spojeny vrcholy a a b do jednoho, pak je nový vrchol vytvořen ve zhrubnutém grafu, a jeho váha je nastavena jako součet vah vrcholu a a b. V případě, že oba vrcholy a a b jsou propojeny hranami s vrcholem c, pak váha nové hrany mezi novým vrcholem $\mathrm{v}(a, b)$ a vrcholem c je rovna součtu vah hran z $\mathrm{h}(a, c)$ a $\mathrm{h}(b, c)$.

K nalezení hran ke zhroucení je použitá metoda *maximální párování*. Výstupem této metody je množina hran ze zadaného grafu, ve kterých se nevyskytuje žádný vrchol dvakrát. Pro maximální párování je také specifické, že už nemůžeme do výsledné množiny přidat žádnou hranu, aniž bychom nabourali podmínku a jedinečnost vrcholů. Hroucením těchto hran se výrazně sníží váhy hran v hrubším grafu, a zlepší se kvalita výsledného rozdělení. Pokud se stane, že nějaký vrchol nebude obsažen v *maximálním párování*, pak se tento vrchol přesouvá do hrubšího grafu nezměněn. Samotný proces zhrubnění končí, jestliže je počet vrcholů v hrubém grafu menší než zadaný práh. Další možností ukončení procesu je případ, kdy snížení množství vrcholů po sobě následujících grafů je menší než nějaký koeficient.

3.5.1.2 Počáteční rozdělení (Initial Partitioning) V této fázi je použita rekurzivní bisekce k rozdělení hrubého grafu. Používá se zde také Kernighan-Lin ke zlepšení kvality rozdělení. Tato část je z časového hlediska ze všech tří části nejméně náročná.

3.5.1.3 Zjemnění (Unoarsening) Jakmile rozdělíme graf pomocí bisekce, je postupně pomocí sekvence zhrubňování grafu získáván původní graf. V tomto procesu se znovu využívá obdoba Kernighan-Lin nazývaná náhodný k-way zjemnění. Vrcholy, které jsou na hranicí, jsou přesunuty do sousedského shluku. A to za předpokladu, že to bude mít za následek redukci rozdělených hran mezi shluky.

Obrázek 8: Ukázka algoritmu pro dělení grafu v Metis [21]



Jak algoritmus v programu Metis funguje, je znázorněno v obrázku 8.

3.5.2 Změny Metisu na Parmetis

Na rozdíl od našeho intuitivního *random walk* je každá ze tří částí *Metisu* předělaná pro používaní MPI knihovny. Graf je předem rozdělen, a každému procesu je náhodně přidělena část grafu.

3.5.2.1 Zhrubnění (Coarsening) Paralelní verze této části se zaobírá hledáním *maximální párování*, a je rozdělena na liché a sudé cykly. Každý cyklus obsahuje dva kroky. V prvním kroku procesy projdou lokálně neoznačené vrcholy v, a pro každý vrchol se snaží nalézt sousední vhodný uzel u pro přidání jejich hrany do množiny *maximální párování*. Při hledání sousedního vhodného vrcholu mohou nastat tři situace. Sousední vrchol u je uložen v lokání části procesu, pak je proces hledaní vhodného vrcholu úspěšně ukončen. Druhý případ: pokud je lichý cyklus $v.váha < u.váha$, nebo je cyklus sudý a platí $v.váha > u.váha$, pak proces pošle požadavek vhodnosti procesu vlastnící vrchol u. Může také nastat třetí případ, kdy podmínky prvního a druhého případu nejsou splněny. Poté je proces nalézání vhodnosti odložen do příštího cyklu. Ve druhém kroku každý proces obdrží požadavky vhodnosti. Procesy vyřeší konflikty libovolně a informují odesilatele o požadavku vhodnosti. Algoritmus pro hledání *maximální párování* končí, pokud byla nalezená vhodnost pro velkou část uzlů. O jakou část se jedná určuje zadaný parametr.

3.5.2.2 Počáteční rozdělení (Initial Partitioning)

V této části je každý kousek grafu rozdělený mezi všechny vlákna používající broadcast komunikaci. Každý proces poté zkoumá jednu větev z rekurzivně bisekčního stromu. Tato rekurzivní bisekce je postavena na *vnořené disekci* [22].

3.5.2.3 Zjemnění (Unoarsening)

Náhodný Kernighan-Lin k-way je paralelizovaný podobně jako hledání *maximální párování*. Algoritmus běží v cyklech. Každý cyklus se skládá ze dvou kroků. V prvním kroku uzly mohou být pouze přesunuty do větších shluků, ve druhém kroku mohou být uzly přesunuty pouze do menších shluků. Tento střídající přístup pomáhá vyhnout se situacím, kde by přesun uzlů do nových shluků lokálně vedl ke snížení počtu řezů, ale globálně ke zvýšení.

3.6 Dělení se znalostí geometrických informací

Doposud se práce věnovala metodám, které neřeší geometrické informace původního zdroje dat. Nyní je prostor věnován krátkému seznámení s případy, kdy nás zajímají souřadnice z původní struktury, které připojujeme k vrcholům do našeho grafu. Tyto algoritmy také předpokládají, že každý vrchol z grafu je fyzicky stejně blízko k ostatním vrcholům jako ve zdroji dat. Zajímavé na nich je také to, že nepracují s informacemi o hranách. Dělení spočívá v rozdělování množiny vrcholů na dvě stejně velké podmnožiny vrcholů. Metody, které využívají geometrické informace jsou tyto: *Coordinate Bisection, Inertial Bisection, Geometric Bisection.* V případě zájmu lze o těchto metodách více zjistit zde: [11], [6], [8].

3.7 Výpočet kvality shluku

Do výsledného programu jsou implementovány tři výpočty kvalit. Jsou to tyto: modularita, konduktance a silhouette index.

První z nich je *modularita*. Modularita je index, který číselně vyjadřuje, zda je graf rozdělen na správné moduly. Je to rozdíl částí hran, které spadají do vytvořených shluků mínus část očekávaných hran, které by spadaly do těchto skupin v náhodném grafu při stejné distribuci stupňů. Hodnota může nabývat hodnot v intervalu < -0.5, 1), kde vyšší hodnota znamená lepší rozdělení na moduly. Vice lze nalézt zde [23]. Je možno použít následující vzorec:

$$Q(C) = \sum_{c \in C} [\frac{|E_{(c)}|}{m} - (\frac{\sum_{v \in C} stupen(v)}{2m})^2]$$

Kde $C = \{C_1, \ldots C_k\}$ je množina shluků, m je počet hran v grafu a $|E_{(C_i)}|$ je počet takových hran grafu, jejich oba vrcholy patří do shluku C_i.

Další z nich, je *konduktance* [24]. Tato hodnota je založena na hranových řezech. Je to poměr velikosti hranového řezu oproti propojenosti uvnitř shluků. Její hodnota nám říká, zda jsme udělali shluky s minimálním počtem řezů. Výsledné hodnoty jsou v intervalu <0,1>, kde čím nižší číslo, tím je řez hran menší.

$$S(C) = \frac{\forall_{c \in C}(\sum_{v \in C} \frac{\sum_{u \in v.sousedi} u \notin C}{stupen(v)})}{C.Pocet}$$

V tomto vzorečku je v čitateli počet vytvořených shluků a ve jmenovateli je součet hodnot reprezentující každý vrchol v grafu. Tato hodnota vrcholu je podíl jeho množství hran vedoucí mimo jeho shluk ke stupni tohoto vrcholu.

Poslední z nich je *silhouette index* [25], který pracuje se dvěma hodnotami. Soudržnost (cohesion) vyjadřující jak blízko si jsou objekty ve shluku, a separace (separation), která měří jak odlišný, nebo jak dobře separovatelný je shluk od ostatních shluků. Jelikož silhouette index je vytvořený pro matice, jejichž ohodnocení hran vyjadřuje vzdálenost vrcholů a ne podobnost, musíme naše normované váhy hran odečíst od jedničky, abychom dostali vzdálenosti. Pro každý vrchol se zjišťuje soudržnost, která je vyjádřena průměrnou vzdáleností daného vrcholu k jeho sousedům uvnitř shluku. Také hledáme jeho separaci. Pro získání této hodnoty vypočítáme průměrnou vzdálenost sousedních vrcholů nacházejících se v jiných shlucích, než je náš aktuální vrchol. Tato hodnota se získává pro každý shluk zvlášť a vybere se pouze ta nejmenší hodnota. Výsledný silhouette index pro daný vrchol je podíl, kde čitatel je rozdíl soudržnosti od separace, a jmenovatel větší číslo z obou hodnot. Abychom získali Silhouette index grafu, spočítáme průměrnou hodnotu SI vrcholů. Silhouette index vrací čísla v intervalu < -1, 1>.

$$s(x) = \frac{sep(x) - coh(x)}{max(coh(x), sep(x))}$$

4 Vlastní implementace

Tato kapitola se bude zabývat vlastní implementací *random walk* algoritmu. S tím se ale pojí samotna implementace reprezentace grafů, kterou bude náš *random walk* používat. A také si zde ukážeme následné převedení sekvenčního algoritmu na *random walk* využívající MPI knihovnu.

4.1 Implementace grafové reprezentace

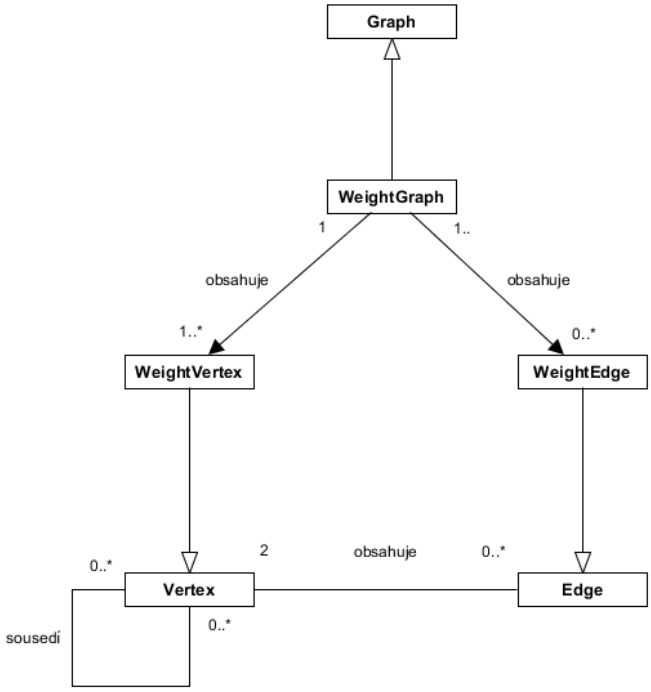
Problematika řídkých maticí již byla probrána v jedné z předešlých kapitol. Byly ukázány různé způsoby, jak matici uchovat v paměti, aby nebyly zbytečně neuchovávány informace o neexistujících hranách, čímž dojde ke snížení potřebného množství paměťi a zároveň urychlení práce s maticí. Jelikož v tomto algoritmu nejsou využity žádné maticové operace jako násobení, součet, rozdíl atd..., nemusí vstupovat do algoritmu samotná maticová reprezentace zadaného grafu. Samotné vrcholy a hrany jsou uchovávány v samostatném datovém seznamu *List*, který je obsažen v API.NET C#. Tyto struktury se nacházejí v abstraktní třídě *Graph*. Třída obsahuje obecnou představu o tom čím by měl být graf tvořen, tedy seznamem hran a seznamem vrcholů. O tom jakého typu (ohodnocené, neohodnocené) budou hrany a vrcholy tvořeny, je ponecháno konkrétně na podtřídě. Samotný vrchol je implementován třídou *Vertex*, která obsahuje vlastnosti: jedinečné identifikační číslo, název, stupeň, referenci na sousední vrcholy a referenci na vlastní hrany. Třída *Edge* zase obsahuje vlastnosti jako: jedinečné identifikační číslo a referenci na incidenční vrcholy. Z uzlu se jde pomoci seznamu *LinkedList* dostat do jakéhokoli sousedního uzlu, a pokud použijeme sousední vrchol jako klíč do struktury *Dictionary*, pak jako výstup dostaneme hranu, která je spojuje. Do třídy *Graf* jsou přidány metody, které nám umožní práci s jeho vrcholy a hranami.

Od třídy *Graph*, *Vertex* a *Edge* dědí specifičtější třídy, které využíváme v našem RW algoritmu. Třída *WeightGraph* dědí od třídy *Graph* a upřesňuje, že budeme pracovat s ohodnocenými vrcholy a grafy (ve skutečnosti nám pro náš algoritmus stačí jen ohodnocené hrany). *WeightVertex* dědí z *Vertex* a obsahuje navíc informace o váze hrany. *Edge* je otcovskou třídou pro *WeightEdge*, která také rozšiřuje hrany o váhu a obsahuje strukturu *Dictionary*, která obsahuje váhy hran získané přepočtem hran z neorientovaných na orientované. Přijímá jako klíč vrchol ze kterého hrana vychází a vrací odpovídající váhu. Váhy hran jsou přepočítávány do normované velikosti. To znamená, že součet všech vah vedoucí z jakéhokoli vrcholu bude 1. Provázaní tříd je zachyceno na obrázku 9.

4.2 Implementace random walku

Pro urychlení výsledného algoritmus je důležité si uvědomit, že zadaný graf nemusí být *souvislý*. Proto jsou nejdříve nalezeny souvislé grafy a pro všechny části, které mají počet vrcholů menší, než je nějaká prahová hodnota, je vytvořeno vlastní RW okolí. Algoritmus pak řeší pouze ty vrcholy, které nejsou po určení *souvislých komponent* přiřazeny do žádného RW okolí. Pro realizaci samotných komponent byla vytvořena třída *Con-*

Obrázek 9: Třídní diagram grafové implementace



nectedComponent, která obsahuje *List* vrcholu, které patří do dané komponenty. Metoda *CreateComponents* hledající tyto souvislé komponenty je ve třídě *WeightedGraph*.

Pro hledaní souvislých komponent byl využit algoritmus 7 pro prohledávání grafu do hloubky. Ten prochází všechny zatím nenavštívené vrcholy, vytvoří pro ně komponentu a do komponenty přidá zbývající vrcholy, které patří do dané souvislé komponenty. Přidávaní zbývajících vrcholu do komponenty probíhá následovně: navštíví vrchol a přidá všechny jeho sousedy, které zatím nebyly nalezeny na zásobník. Ze zásobníku poté postupně vybírá vrcholy a rozbaluje jeho sousedy. To probíhá tak dlouho, dokud zásobník nebude prázdný.

4.2.1 Vytváření RW okolí

Samotný *random walk* algoritmus přebírá grafovou reprezentaci, kterou byla popsána výše jako vstupní parametr. Mezi další parametry patří počet kroků v jednotlivých sledech, minimální počet vytvořených sledů při tvorbě RW okolí, minimální procentuální překryv pro sjednocení RW okolí a zda v algoritmu se má využívat hodnota návštěvnosti. Samotný sekvenční algoritmus lze najít ve třídě *RWPartitioningAlgorithm* a klíčová je metoda *DoAlgoritm*. Při zavolání této metody se spustí hledání RW okolí.

V metodě *DoAlgoritm* je možné vidět cyklus, který nekončí dokud existují vrcholy, které nebyly přiřazeny do žádného RW okolí. Uvnitř cyklu se zjišťují počáteční vrcholy, ze kterých se vytvoří jednotlivé sledy vedoucí ke RW okolím. Jako datová reprezentace RW okolí je použita třída *RWCluster* (v aplikaci se nerozlišuje RW okolí a shluk), která

```
Vstup: graf G = (V, H);
Výstup: souvislé komponenty K_1 = (V_1, H_1), ....K_n = (V_n, H_n);
for pro každý vrchol do
    if vrchol nenáleží do komponenty then
        vytvořit komponentu;
        vložit vrchol na zásobník;
        while zásobník není prázdný do
            získat aktuální vrchol ze zásobníku;
            přiřadit aktuální vrchol k nové komponentě;
            for každý sousedící vrchol s aktuální vrcholem do
                if pokud sousedící vrchol nepatří do komponenty a zároveň není na
                zásobníku then
                    vložit sousedící vrchol na zásobník;
                end
            end
        end
    end
end
```

Algoritmus 7: Prohledávání grafu do hloubky

obsahuje informace o jednotlivých sledech a hodnoty návštěvnosti jednotlivých vrcholů, které jsou reprezentovány v *Dictionary*, kde klíč je samotný vrchol. *RWCluster* také uchová své identifikační číslo a zda je vytvořený při hledání souvislých komponent. Cyklus volá metodu *CreateRWpath*, jejíž úkolem je vytvořit sledy a přidat je do nového RW okolí. Samotné vyhodnocení vrcholů, které nakonec skončí v RW okolí, si vyhodnocuje třída *RWCluster* sama. Algoritmus pro výběr následujícího vrcholu ve sledu je implementován v metodě *MoveToNextVertex*. V této metodě je tudíž dříve popsaný algoritmus Rulety pro náhodný výběr. Jakmile nastane situace, že všechny vrcholy jsou přiřazeny do RW okolí, pak poslední metodu kterou *DoAlgoritm* zavolá je metoda *SolveConflicts*, kde se řeší překrývající RW okolí.

4.2.2 Slučování RW okolí

Pro počáteční část řešení konfliktů, slučování RW okolí, je vytvořena metoda *MergePartitions*, která je nejpomalejší částí algoritmu. Po tomto zjištění následovala snaha vytvořit rychlejší způsob slučování RW okolí. Z toho důvodu se následující způsob nemusí zdát z počátku příliš efektivní.

Pro každé RW okolí *RWCluster*, které se nachází v *List*, jako vlastnost třídy *RWPartitioningAlgorithm*, projdeme všechny následující RW okolí v seznamu a pro každé z těchto RW okolí spočteme společné vrcholy. A pokud počet společných vrcholů je větší než zadaná minimální procentuální shoda, pak jsou tyto RW okolí sloučena. Na tamto algoritmu je zajímavé, že vybere pro sloučení první RW okolí v seznamu, které vyhovuje podmínce sloučení. Ale to samozřejmě neznamená, že to je RW okolí s největším procentuálním pře-

kryvem. Toto procházení RW okolí končí ve chvílí, kdy projdeme celý seznam RW okolí a nepodaří se žádné RW okolí sloučit.

Další možností je, že pro každé RW okolí, na základě jeho vrcholů, zjistíme RW okolí, které se s daným RW okolí překrývá nejvíce. Pak se pokusíme nově vytvořené RW okolí sloučit s nějakým jiným RW okolím. Slučování také končí ve chvíli, kdy se nepodařilo po celém průchodu žádné RW okolí sloučit. Tento algoritmus se nakonec ukázal, jako pomalejší, protože se odstranila jistá náhodnost při výběru RW okolí ke sloučení a nyní bylo vybíráno to nejvíce vyhovující. Vznikalo nám tedy jedno velký RW okolí a velkém množství vrcholů, ke kterému se přidávali ostatní. Na místo toho v původním algoritmu vznikaly přibližně rovnoměrně rostoucí části, které se nakonec sjednotily.

4.2.3 Řešení zbývajících konfliktních vrcholů

Posuneme se tedy dále k řešení zbývajících konfliktních uzlů, které je implementované ve dvou metodách *DistributionOfVertexByNS* a *DistributionOfVertex*. Která z nich se zavolá, závisí na zadaném parametru, který určuje, zda jsou používány hodnoty návštěvnosti.

DistributionOfVertex řeší problematiku přidání konfliktního vrcholu k RW okolí pomocí množství výskytů vrcholů ve sledech, které obsahuje samotné RW okolí. Algoritmus spočívá v tom, že projde všechny vrcholy v cyklu a pokud se nachází ve více, než jednom RW okolí, zjistí jeho výskyt ve sledech a přiřadí vrchol RW okolí, v jehož sledech se objevil nejčastěji. Pokud tento princip nepomůže, vytvoří se pro vrchol pseudo RW okolí a pokud ani to nepomůže, vybereme výsledný RW okolí náhodně, tak jak jsme algoritmus již popsali.

Druhá možnost v podobě *DistributionOfVertexByNS* prochází vrcholy také v cyklu a pro každý konfliktní vrchol zjišťuje jeho hodnotu návštěvnosti ze všech RW okolí do kterých patří. Pro tu s největším hodnotou se zachová a pro ostatní se odstraní.

4.2.4 Korekce konfliktních vrcholů

Po implementaci těchto rozhodujících algoritmů, ale vznikly problémy, se kterými se v původním návrhu neuvažovalo. Zdrojový kód řešení těchto problému je k nalezení v metodě *CorrectionOfDistributionAlgorithm*. První z nich vystával z důvodů počátečních vrcholů ve svých shlucích, se kterými nemůže konkurovat výskyt tohoto vrcholu v jiném shluku. Proto se občas stávalo, že pokud si vrcholy ze shluku rozdělily jiné shluky, pak tento počáteční vrchol zůstával ve svém shluku sám. Jelikož už jsou všechny vrcholy přiřazeny do shluku, můžeme našemu vrcholu zjisti sousední shluk, do kterého by se na základě ohodnocení jeho hran hodil nejlépe. Můžeme tedy pomoci metody *GetVertextToBestPartitionByWight* zjistit, ke kterému jinému sousednímu shluku náš zbývající počáteční vrchol ve svém shluku připadá nejlépe. Metoda pro zadaný vrchol prochází sousedy a zjišťuje v jakém shluku se sousední vrchol nachází. A ve struktuře *Dictionary* ukládá pod klíčem, který je identifikátor sousedního shluku, součet vah hran vedoucí do sousedního shluku. Pokud se vyskytnou shluky se stejným součtem vah, pak vrchol přidáme k některé z nich náhodně.

Druhý problém může nastat v případě hraničního uzlu, který spadal pod dva RW okolí a jeho jediný sousedící vrchol taky spadá pod dva RW okolí. Na základě našich

rozhodovacích algoritmů, ale každý z těchto sousedních vrcholů byl korektně přiřazen pod jiný shluk. V toto případě vzniká problém, protože v jednom shluku vznikl ostrůvek, ke kterému se z jeho shluku nevede žádná cesta. Proto je nutné projít každý vrchol a kontrolovat zda má ve svém shluku aspoň jeden sousedící vrchol. Pokud ne zavoláme zase metodu *GetVertextToBestPartitionByWight*, která přiřadí vrchol do nejlepšího sousedícího shluku. Oba tyto cykly můžete vidět v tomto algoritmu 8.

Vstup: shluky $S_1 = (V_1, H_1), ...S_n = (V_n, H_n)$, kde může nastat osamělý samotný počáteční vrchol, nebo vrchol ve shluku, který nemá ve shluku žádného souseda;
Výstup: shluky $S_1 = (V_1, H_1), ...S_n = (V_n, H_n)$ kde nemůže nastat osamělý samotný počáteční vrchol, nebo vrchol ve shluku, který nemá ve shluku žádného souseda;
for *pro každý shluk* **do**
  if *shluk obsahuje jenom jeden vrchol a tento shluk nevznikl při hledání souvislých komponent* **then**
    Přidat vrchol do nejlepšího sousedního shluku na základě vah hran;
  end
end
for *pro každý vrchol* **do**
  if *vrchol nemá ve shluku žádný sousední vrchol a shluk, ve kterém se nachází, nevznikl při hledání souvislých komponent* **then**
    Přidat vrchol do nejlepšího sousedního shluku na základě vah hran;
  end
end

Algoritmus 8: Korekce rozhodovacíh algoritmu

4.3 Implementace random walku využívající MPI knihovnu

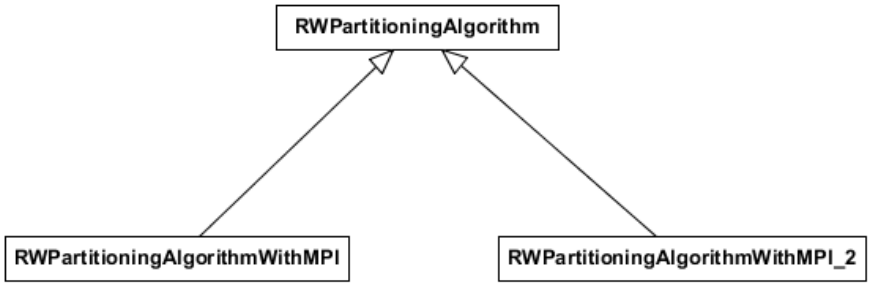
Implementace random walku využívající MPI knihovnu byla vytvořena ve dvou verzích. Obě verze dědí od sekvenčního algoritmu. Dědičnost je znázorněná zde 10. Jejich implementace je možné najít ve třídě *RWPartitioningAlgorithmWithMPI* a *RWPartitioningAlgorithmWithMPI_2*. V *RWPartitioningAlgorithmWithMPI* je implementovaná verze pro vytváření jednoho RW okolí s využití všech výpočetních uzlů. Zato *RWPartitioningAlgorithmWithMPI_2* využívá každý výpočetní uzel, kromě řídicího, k tvorbě vlastního RW okolí.

Pro urychlení algoritmu je možné využití parametru pro načítání grafu ze společného úložiště na síti, kde každý z výpočetních uzlu si načte graf samostatně. Pokud jsou použity stejně silné výpočetních jednotky, zabere tato operace na každém uzlu přibližně stejnou dobu. V opačném případě graf načte výpočetní uzel a rozešle jej ostatním pomoci MPI metody *Broadcast*, což může trvat nezanedbatelný čas v závislosti na velikosti grafu.

4.3.1 Random walk s MPI vytvářející společně jedno RW okolí

V případě využití tohoto algoritmu je důležité si dát pozor na to, aby počet zadaných sledů nebyl menší než počet výpočetních uzlů. Rozdělení sledů mezi uzly by nám dávalo číslo

Obrázek 10: Dědičnost random walk algoritmu



menší než 1, což by vedlo k havárii programu. Tento chybný stav je ale ošetřen chybovým hlášením, takže se program s těmito parametry nepovede spustit.

Program v metodě *DoAlgoritm* je rozdělen do dvou větví. Větev pro řídicí uzel a větev pro ostatní. Uzly které nejsou řídicí okamžitě vstoupí do metody *CreateRWpath*, kde čekají na zprávu od řídicího uzlu, která udávající počet sledů, které mají vytvořit pro nový RW okolí. Toto číslo přijde pomoci metody *Broadcast*. Pokud hodnota iterace bude -1, pak výpočetní uzel ví, že se má ukončit. V jiném případě očekává přijetí *ID* počátečního vrcholu a podle něho si uzel zjistí samotný objekt *Vertex* a následně pomoci metody *CreateRWpathByRank* vytvoří požadované množství sledů. Vytvořené sledy navrátí pomoci metody *Gather*. V případě, že je vybrána verze algoritmu pro prací s hodnotami návštěvnosti, je výsledek posílán jako struktura *Dictionary*, kde klíčem je identifikátor vrcholu a hodnota je návštěvnost vrcholů. Pokud je vyberán algoritmus využívající výskyty ve sledech, pak se posílají seznamy *List*, kde každý list reprezentuje jeden sled. Cyklus se potom opět spustí a čeká se příjem počtu sledů.

Pokud je ale aktuální uzel řídicí, z *DoAlgoritm* se dostaneme do metody *WorkLikeRoot*, kde stejně jako v sekvenčním algoritmu je cyklus, který běží tak dlouho dokud existují vrcholy, které nemají žádný přiřazený RW okolí. V tomto cyklu pokaždé vytvoří novou prázdnou instanci RW okolí a pak se ji snaží naplnit. Učiní tak pomoci metody *CreateRWpath*, kde posílá počáteční uzel jako parametr. A v této přetížené metodě určí kolik sledů má jednotlivý výpočetní uzel vytvořit a tuto hodnotu rozešle ostatním výpočetním uzlům. Pošle také ID počátečního vrcholu. Výsledné sledy pak příjme pomoci metody *Gather*.

Způsob zpracování sledů se zase liší podle toho, jaký algoritmus byl zvolen. V případě varianty odvozené od *Neighbourhood Similarity*, je přijato pole struktur *Dictionary*, které je převedeno na jednu strukturu *Dictionary*. Klíčem je samotný objekt *Vertex* a ne jenom jeho ID. Všechny přijaté hodnoty návštěvnosti pro daný vrchol se sečtou a uloží se do struktury pod odpovídající vrchol. Výsledný *Dictionary* se přidá do nového RW okolí. Při využití druhé možnosti v podobě výskytu ve sledech, přijme od každého výpočetního uzlu pole seznamu typu *List*. Tyto seznamy od každého uzlu jsou sjednoceny do jednoho seznamu *List*. Takže vznikne dvou úrovňové pole *Listu*.

V obou dvou případech je určeno, které vrcholy ze struktury budou přidány do RW okolí. Práce s určováním počtu iterací je stejná, jako u sekvenčního algoritmu. Po ukončení

algoritmu řídicí uzel rozešle metodou *Broadcast* hodnotu -1, aby informoval ostatní výpočetní uzly, že mají ukončit své cykly.

4.3.2 Random walk s MPI vytvářející každý jeden RW okolí

Pokud je tedy zadán parametr, kterým určíme použití algoritmu využívajícího MPI knihovnu s tím, že pro každý uzel se bude vytvářet vlastní RW okolí, pak se nám spustí *DoAlgoritm* v *RWPartitioningAlgorithmWithMPI_2*. Tato metoda stejně, jako v minulém případě, zavolá odpovídající metodu pro odpovídající výpočetní uzel. V případě, že se jedná o řídicí uzel, pak je zavolán *WorkLikeRoot* a ve všech opačných případech dochází k přechodu do *CreatePartitions*. Neřídicí výpočetní uzly po zavolání *CreatePartitions* vstoupí do cyklu, ve kterém v každém průchodu vytvoří nové prázdné RW okolí a pomoci metody *Scatter* získají od řídicího uzlu ID počátečního vrcholu. Po zjištění odpovídajícího vrcholu, v podobě *Vertex* třídy, spustí vytváření nového RW okolí pomoci metody *CreateRWpath* stejným způsobem, jako sekvenční algoritmus. Ve variantě využívající hodnoty návštěvnosti pošleme pomoci metody *Gather* strukturu *Dictionary*, která obsahuje stejná data, jak v *RWPartitioningAlgorithmWithMPI*. A to identifikátor vrcholu, jako klíč a jako odpovídající hodnotu pod klíčem najdeme hodnotu návštěvnosti daného vrcholu. V případě využívající variantu výskytu vrcholu ve sledech pošleme metodou *Gather* seznam *List*, ve kterém se nacházejí výsledné vrcholy pro nové RW okolí. Práce neřídicích uzlů končí, pokud obdrží zadaných identifikátor vrcholu -1.

Zatím co řídicí uzel v metodě *WorkLikeRoot* vytvoří nejdříve souvislé komponenty a odpovídající shluky k těmto komponentám. Před začátkem cyklu zjistí odpovídající množinu počátečních uzlů a rozešle je pomoci metody *Scatter*. Pak vstoupí do cyklu, který běží tak dlouho, dokud existuje nějaký vrchol nepřiřazený RW okolí. Okamžitě po vstoupení do cyklu zjistí další množinu počátečních vrcholů a rozešle je stejným způsobem. A pak teprve přijímá metodou *Gather* vytvořené konečné RW okolí. Odeslání počátečních vrcholu na začátku algoritmu je z důvodu ušetření času řídicího uzlu při čekání na vytvořené RW okolí. Po ukončení cyklu přijmeme poslední množinu a odešleme pro každý pomocný výpočetní uzel -1 pomoci metody *Scatter*. Pomocné výpočetní uzly se ukončí.

4.4 Výsledek implementace

Na základě toho, jaké parametry jsou při spuštění zvoleny, se spustí různé verze algoritmus. Pro určení kvality shluků jsou spočítány tři hodnoty kvalit a samozřejmě dostaneme výsledné shluky a to ve formátu *gdf*, kde každý shluk má náhodně přidělenou barvu. V této části jsou tedy vysvětleny výpočty kvality a jsou ukázány výsledky algoritmu. Mnohem více výsledku ale lze ale najít v následující kapitole věnované experimentům.

Výsledný program jménem *GraphPartitioning* lze pomoci příkazové řádky spustit více rozdílnými způsoby. Pokud je spuřtěn samotný *GraphPartitioning*, je nutné zadat minimálně parametry -i a -o, které slouží pro vstupní soubor a pro výstupní soubor.

Vstupní soubor je očekáván v GDF formátu:
vrchol1 vrchol2 váhaHrany1
vrchol3 vrchol4 váhaHrany2

vrchol5 vrchol6 váhaHrany3

Kde každý řádek symbolizuje hrany mezi vrcholy a poslední hodnota je váha těchto hran. Pokud vstupní soubor neobsahuje váhy hran, je váze přiřazena hodnota 1. Pokud je na řádku pouze samostatný vrchol, znamená to, že vrchol v grafu existuje, ale nemá řádné hrany.

Výstupní soubor je také ve formátu GDF, ale obsahuje přiřazení barev jednotlivým shlukům.

Dalšími parametry, které je možné zadat jsou -k a -t, kde číslo zadané za parametrem -k je počet kroků v jednotlivých sledech a parametr -t udává minimální počet vytvořených sledů při vytváření RW okolí. Pokud tyto parametry nejsou zadány, nezpůsobí to pád programu, ale oběma se přiřadí číslo 5. Další důležitý parametr je -p ve kterém se udává jakou minimální procentuální shodu musí mít RW okolí, aby je bylo možné sloučit. Hodnota parametru -p se zadává v jeho desetinné podobě, kde 0,70 znamená 70%. Výchozí hodnota parametru -p je 0,40. Tyto parametry se zadávají ať vybereme jakýkoli algoritmus. Pokud je ale spuštěn pouze sekvenční algoritmus, pak ještě můžeme použít parametr -s, který udává, zda algoritmus má používat hodnoty návštěvnosti.

Následující ukázkou je spuštěn výsledný sekvenční algoritmu pro dělení grafů, kde zadaný graf je v souboru *test* a výstupní v *otest*. Můžeme si všimnout, že budeme využívat hodnoty návštěvnosti a minimální procentuální shoda vrcholů pro sjednocení RW okolí je 49 %. *GraphPartitioning -i test -o otest -s -p 0,49*

V případě použití MPI knihovny, kdy je výsledný program volán způsobem, že je nejdříve zadán příkaz mpiexec a k němu paprametr -n, udávající kolik výpočetních uzlů se má MPI použít a až pak náš program s potřebnými parametry. Program pozná, že je spuštěn pro více výpočetních uzlů a pokud je zadán parametr -c, spustí se algoritmus, kde každý RW okolí je tvořen na vlastním jádře. Pokud parametr není zadán, spustí vytváření každého RW okolí pomoci všech uzlů. Další pomocný parametr je -a. Ten určí, že každý uzel si načte vstupní graf sám. Parametr -s je je v tomto případě zbytečný, protože algoritmy s vyžitím MPI knihovny je vždy zapnutý.

mpiexec -n 4 GraphPartitioning -i test -o otest -p 0,35 -c -a

V ukázce, jak spouštět algoritmus pomoci MPI na čtyřech uzlech, bude každý uzel načítat vstupní graf ze souboru *test* a po nalezení RW okolí pomocí algoritmu, kde se každé RW okolí vytváří na vlastní jádřem, se RW okolí sloučí s pravděpodobnosti 35%. Nakonec se výsledek uložil do souboru *otest*. Algoritmus je spuštěn s výchozími hodnotami parametrů pro počet kroků ve sledů, a také pro počet sledů v každém RW okolí.

5 Výsledky experimentů nad různými datovými kolekcemi

Tato část práce je určena různým experimentům, které ukazují možnosti výsledné aplikace. Porovnává jeho různé algoritmy mezi sebou jak časově tak kvalitně pro různé zadané parametry.

5.1 Malé grafy

Pro počáteční testování, byly vytvořeny malé pomocné grafy, které měly za úkol ověřit, zda algoritmus opravdu shlukuje dle naší představy. Zároveň byla také vyvinuta snaha nalézt složité případy, na které může algoritmus narazit ve větších grafech. Na grafu 11 lze vidět dva vrcholy, které výpočet algoritmu ztíží. První z nich je vrchol s číslem 3, do kterého vede pět hran. Je patrné, že algoritmus rozdělil graf tak, že třetí vrchol je členem oranžového shluku, do kterého vedou dvě hrany, a nikoli modrého shluku, do kterého vede jedna hrana, což by způsobilo větší řez hran. Další složitý vrchol má číslo 5, do tohoto vrcholu vedou dvě hrany, jedna z modrého shluku a druhá z oranžového shluku. V tomto případě není důležité, do kterého shluku se přidá, protože hranový řez bude stejný. V tomto případě je přidán do modrého shluku, což způsobí, že počet vrcholů v modrém a oranžovém shluku je roven. Na základě toho, jak jsme si algoritmus definovali, lze prohlásit, že je deterministický. Což je způsobeno více důvody, jeden z nich, a také ten nejdůležitější je, že počáteční vrcholy se vybírají náhodně, tudiž se po každém spuštění programu vytvoří různé RW okolí, se kterými se nadále pracuje. Dalším důvodem je samotný způsob slučovaní RW okolí, který, jak je již definován výše, nevybírá pro sloučení to RW okolí se kterým se překrývá nejvíce, ale sloučí se s první RW okolí, které splňuje podmínky sloučení. Jak již bylo uvedeno, důvodem je, že tento způsob se ukázal nejefektnější. Algoritmus tudíž vrací i různě kvalitní výsledky pro stejně zadané parametry. V tomto prvním případě 11 byla zadána pravděpodobnost slučování RW okolí 45 %, počet kroků ve sledu 5 a minimální počet sledů také 5. Výsledky shlukovaní jsou tyto 1, pod pokusem číslo 1.

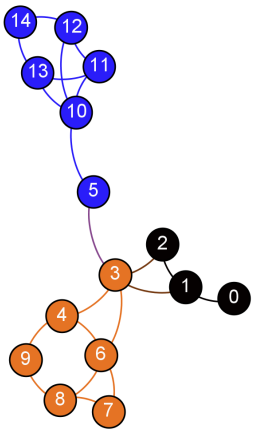
pokus č.	Konduktance	Modularita	Silhouette index
1	0,18	0,46	0.733
2	0,15	0,51	0.733

Tabulka 1: Kvalitativní výsledky ukázkového grafu

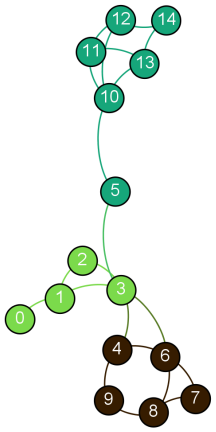
Na obrázku 12 je znázorněn výsledek, který získáme opakovaným spuštěním tohoto programu se stejně zadanými parametry. Lze vidět, že vrchol číslo 3 se přidal do shluku k vrcholům 0, 1, 2, přičemž hranový řez zůstal stejný. Pozměnil se pouze počet vrcholů ve shlucích. Kvality dopadly takto 1, pod pokusem číslo 2.

Jelikož se konduktance snížila a modularita zvýšila, je možné říci, že druhé shlukování je kvalitnější než to první. Je to díky tomu, že se množství vrcholů ve shlucích více vyrovnalo. Místo původního rozložení vrcholů 6, 6, 3, máme rozložení 6, 5, 4.

Obrázek 11: Výsledky rozdělení ukázkového grafu, algoritmu RW, první pokus



Obrázek 12: Výsledky rozdělení ukázkového grafu, algoritmu RW, druhý pokus

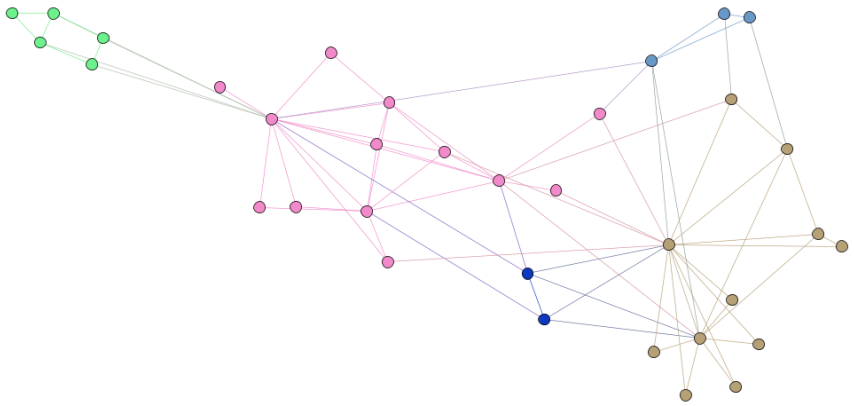


5.2 Porovnávání algoritmů na výskyt ve sledech s hodnocením návštěvnosti

Jak již bylo uvedeno v popisu algoritmu, pro rozhodování jaké vrcholy skončí ve výsledném RW okolí z vytvořených sledů, jsou použity dva různé algoritmy. První z nich počítá výskyty vrcholů ve sledech a druhý používá hodnotu návštěvnosti převzatou z algoritmu *Neighbour Similarity*. Vzhledem k tomu, že bylo v minulém experimentu dokázáno, že algoritmus není deterministický, je tedy algoritmus spuštěn desetkrát pro každý algoritmus. Jejich zprůměrované výsledky kvalit jsou porovnány a výsledky ukazují, který algoritmus je kvalitnější.

Pro testování byl vybrán graf *Zachariho karate klubu*, který představuje sociální síť přátelství, která byla zpřístupněna pro tyto testovací účely. Graf má 34 vrcholů, lze tedy předpokládat, že tolik členů měl také klub, a 78 hran, které reprezentují vztahy mezi členy klubu. Ukázka tohoto grafu lze vidět na tomto obrázku: 13.

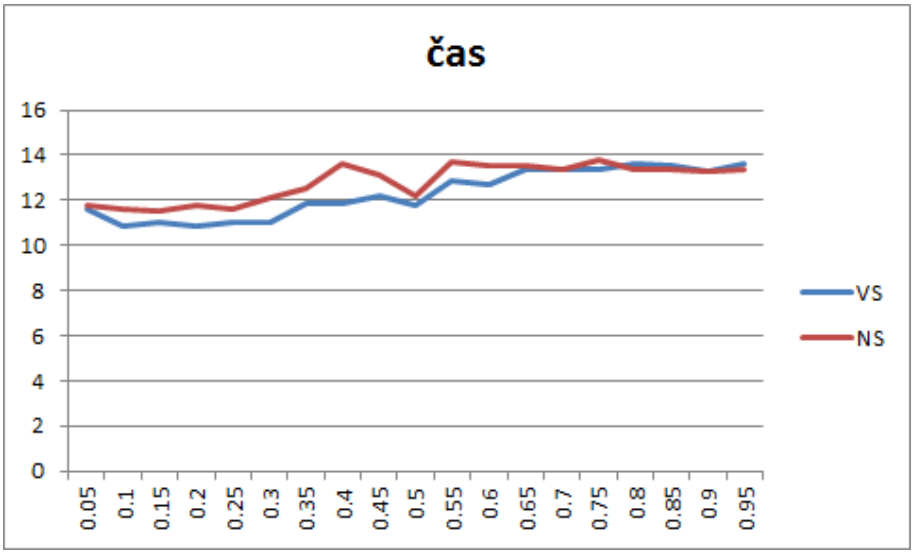
Obrázek 13: Ukázka rozdělení Zachari karate klubu



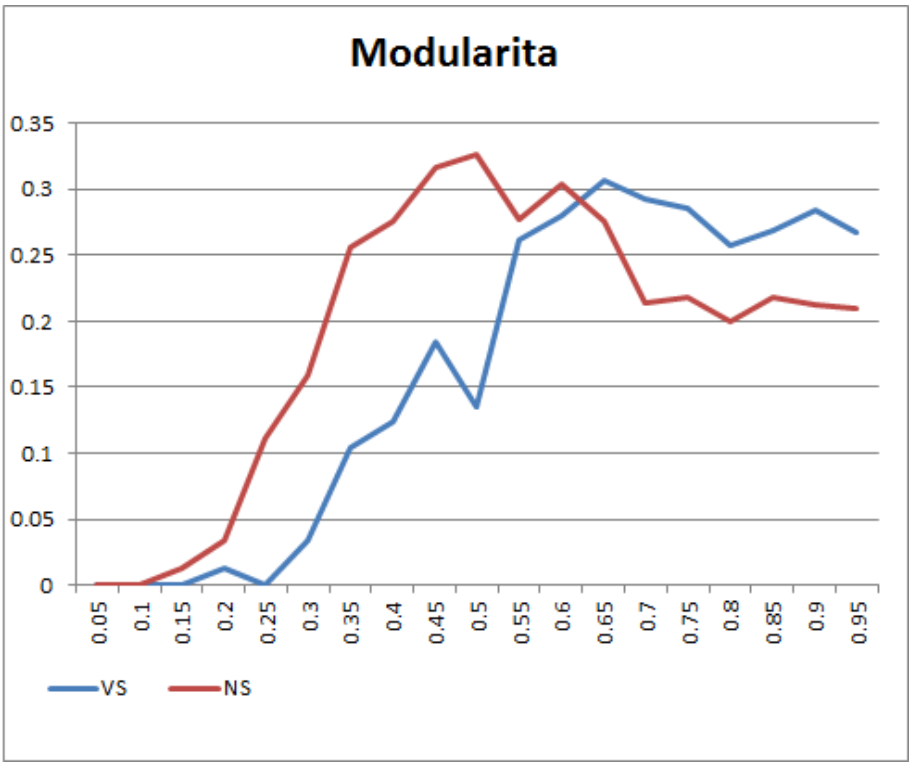
Pro objektivní porovnání těchto dvou algoritmů (výskyty ve sledech a hodnocení návštěvnosti) byl pro zvyšující se parametr minimálního procentuálního překryvu pro sloučení RW okolí spouštěn program vždy desetkrát pro stejné ohodnocení parametru. Výsledné hodnoty kvalit byly pro toto ohodnocení zprůměrovány. Parametr sjednocení začal na hodnotě 5 % a po spuštění deseti nezávislých pokusů pro tuto hodnotu se přičetlo dalších 5 %, a znova se spustilo 10 pokusů. Toto se opakovalo, dokud parametr sjednocení nedosáhl 95 %. Celý tento proces byl spuštěn dvakrát, jednou s parametrem pro využití výskytu ve sledech (VS), a podruhé pro používaní hodnoty návštěvnosti (NS). Výsledek je znázorněn v následující tabulce. Na základě náhodných pokusů byla hodnota parametrů pro minimální počet sledů a počet kroků stanovena na pět. V tabulce 2 vidíme výsledek tohoto experimentu, kde sloupec p představuje parametr pravděpodobnosti udávající minimální procentuální překrytí RW okolí pro jejich sjednocení. Pod *kon.* se neskrývá nic jiného, než konduktance a *mod.* znamená modularitu, ostatní zkratky nepředstavují nic nového.

Výsledky z tabulky 2 jsou zobrazeny v následujících čtyř tabulkách, které porovnávají

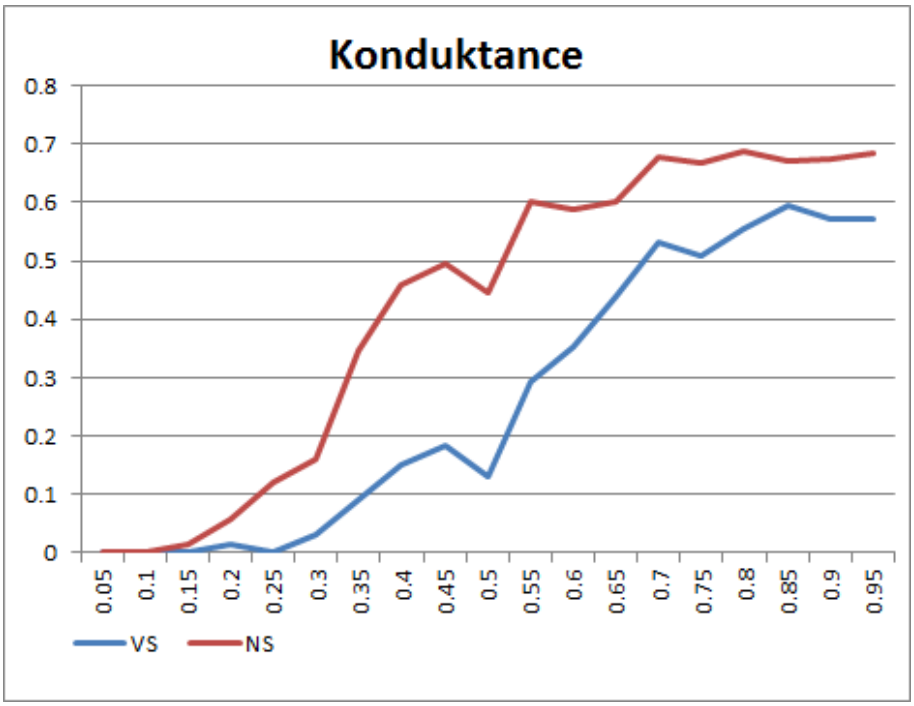
Obrázek 14: Výsledky testu VS vs. NS v čase (milisekundách)



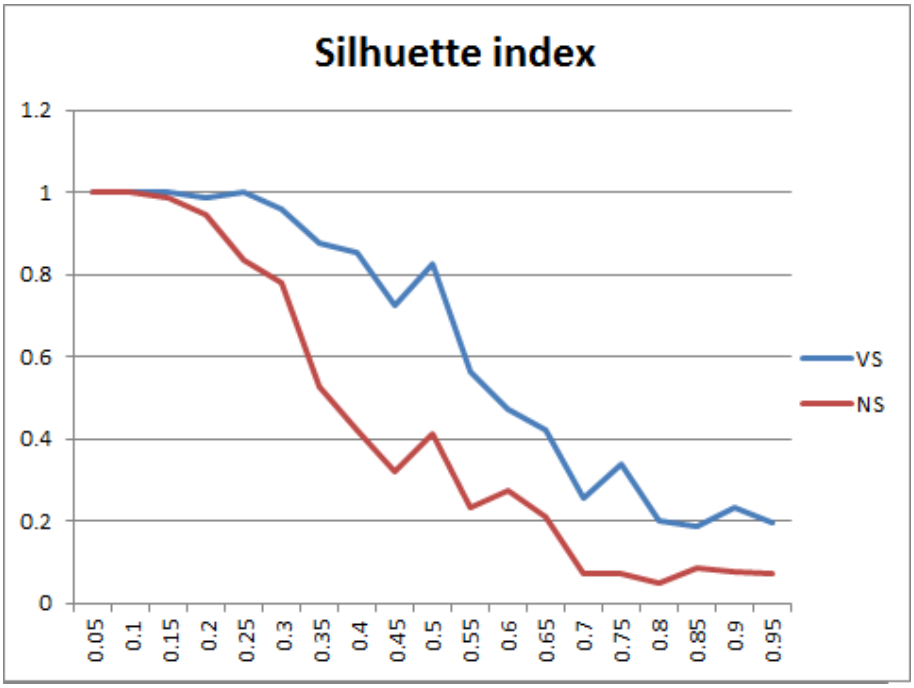
Obrázek 15: Výsledky testu VS vs. NS v modularitě



Obrázek 16: Výsledky testu VS vs. NS v kodukdance



Obrázek 17: Výsledky testu VS vs. NS v silhouette indexu



p	čas VS	čas NS	kon. VS	kon. NS	mod. VS	mod. NS	SI VS	SI NS
0,05	11,60	11,80	0,00	0,00	0,00	0,00	1,00	1,00
0,10	10,90	11,60	0,00	0,00	0,00	0,00	1,00	1,00
0,15	11,00	11,50	0,00	0,01	0,00	0,01	1,00	0,99
0,20	10,90	11,80	0,01	0,06	0,01	0,03	0,99	0,94
0,25	11,00	11,60	0,00	0,12	0,00	0,11	1,00	0,84
0,30	11,00	12,10	0,03	0,16	0,03	0,16	0,96	0,78
0,35	11,90	12,50	0,09	0,35	0,10	0,26	0,88	0,53
0,40	11,90	13,60	0,15	0,46	0,12	0,28	0,85	0,42
0,45	12,20	13,10	0,18	0,50	0,18	0,32	0,73	0,32
0,50	11,80	12,20	0,13	0,44	0,14	0,33	0,83	0,41
0,55	12,90	13,70	0,29	0,60	0,26	0,28	0,56	0,24
0,60	12,70	13,50	0,35	0,59	0,28	0,30	0,47	0,28
0,65	13,40	13,50	0,44	0,60	0,31	0,28	0,42	0,21
0,70	13,40	13,40	0,53	0,68	0,29	0,21	0,26	0,07
0,75	13,40	13,80	0,51	0,67	0,29	0,22	0,34	0,07
0,80	13,60	13,40	0,56	0,69	0,26	0,20	0,20	0,05
0,85	13,50	13,40	0,60	0,67	0,27	0,22	0,19	0,09
0,90	13,30	13,30	0,57	0,68	0,28	0,21	0,24	0,08
0,95	13,60	13,40	0,57	0,68	0,27	0,21	0,19	0,07

Tabulka 2: Výsledky testu mezi VS a NS

čas 14, modularitu 15, konduktanci 16 a silhouette index 17 mezi algoritmy VS a NS. Z výsledků lze vyčíst, že co se týče času, jsou algoritmy velice vyrovnané. Avšak co se týče ostatních hodnot, lze vidět, že pro nižší parametr procentuálního překryvu pro sjednocení RW okolí jsou kvality nejlepší pro silhouette index a konduktanci. Velmi zajímavé je to, že modularita do určité doby stoupá. Z těchto výsledků nelze tedy říci, který parametr se v daném algoritmu chová nejlépe. Je to vždy kompromis mezi SI a konduktanci oproti modularitě. To stejné můžeme prohlásit o algoritmech NS a VS, protože modularita dopadla lépe pro NS ale ostatní dva parametry dopadly lépe pro VS. Pokud bychom ale chtěli rozumný kompromis, tak jako lepší vybereme VS, protože rozdíl nejlepších modularit obou algoritmů není až tak velký.

5.3 Porovnání rychlosti a kvality RW algoritmu

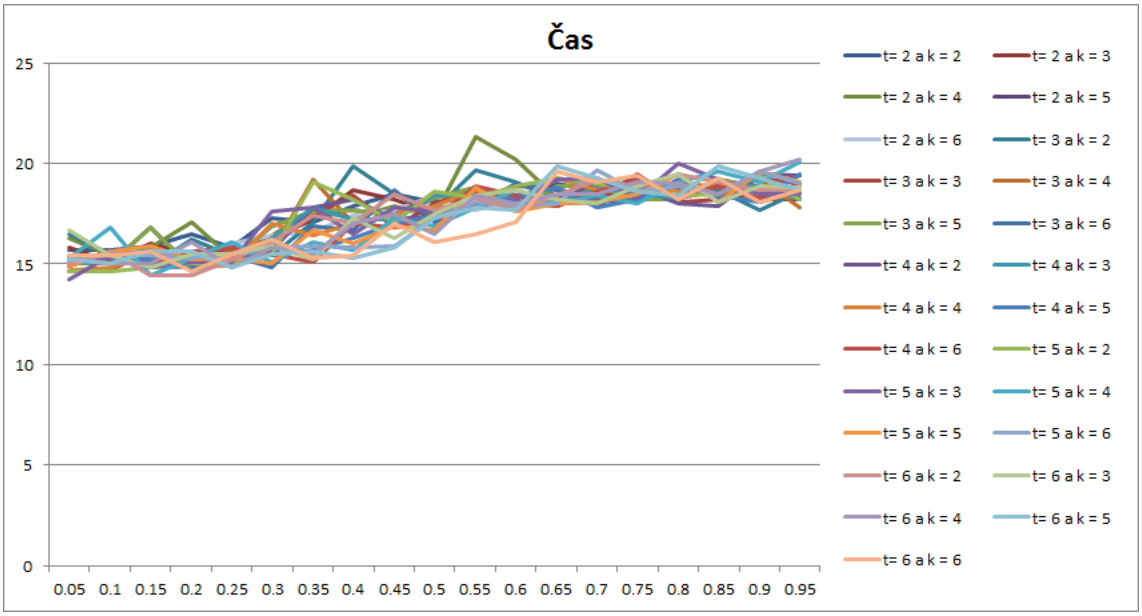
V této části jsou pro testovací graf *Zachariho karate klub*, který není nadměrně velký (ale zároveň není příliš malý) spouštěny různě zadané parametry, aby bylo zjistěno pro jaké zadané vstupní parametry se nám vrátí nejlepší možný výsledek. V úvahu přichází však také varianta, že nejlepší výsledek není nalezen, protože možnost zadání například parametru slučování vrcholů, umožňuje ovlivnit, kolik výsledných shluků je vyžadováno. Pokud tedy parametr bude nižší, je třeba výsledných shluků očekávat více, naopak je-li

zadán parametr vyšší, výsledných shluků bude více. Toto však samozřejmě ovlivní výsledné parametry kvality.

I přesto, že v předešlé kapitole nebylo možné určit, který z algoritmů NS nebo VS je lepší, neboť jejich výsledné hodnoty jsou velice podobné, je tato část určena testování algoritmu VS. Stejně jako v minulé kapitole, je postupně zvyšována hodnota pravděpodobnosti pro slučování RW okolí, a pro každou hodnotu je provedeno deset experimentů. Tento cyklus provádíme jak pro různé parametry počtu sledů, tak pro počty kroků. Oba tyto parametry se budou pohybovat od 2 do 6.

V popiscích křivek se vyskytuje parametr k, který znázorňuje počet kroků v RW okolí, a parametr t, který určuje minimální počet sledů pro startovní vrchol. Parametr t příliš výsledné křivky neovlivní, a to z důvodu, že jeho hodnota se používá pouze pokud je jeho hodnota větší než stupeň počátečního vrcholu. Nejdříve jsou zde uvedeny grafy se všemi experimenty, které na první pohled nejsou zcela čitelné, ale následně jsou vybrány pouze zajímavé výsledky.

Obrázek 18: Výsledky testování času v milisekundách



Co se tyče výsledku času 18, nelze pozorovat nějakou výraznou závislost na zadaných parametrech. Čas se pouze zvyšuje od 15 milisekund po 20 milisekund v závislosti na pravděpodobnosti sjednocení. Důvodem je to, že čím méně RW okolí sjednotíme, tím více je nutno řešit konfliktní vrcholy nacházející se ve více RW okolích.

Dále se podíváme na výsledky konduktance 19, tyto výsledky jsou již oproti času více zajímavé. Z počátečního grafu byly vytvořeny následující tři grafy, které jsou více přehledné, dojde na nich k popisu závislosti parametru k a t. Již v předešlém testování byly vybrány výchozí parametry k = 5 a t = 5. Pro tyto parametry byly vytvořeny dva

grafy, z nichž je jeden parametr statický, a ten druhý nabývá hodnot od dvou do šesti. První znázorňuje změnu parametru počtu iterace 20, druhý znázorňuje změnu parametru počtu kroků 21 ve sledu.

Lze si všimnout, že změny způsobené parametrem k se projeví více, než změny způsobené parametrem t. Čím větší stupeň budou mít vrcholy, tím méně se parametr t projeví, protože se místo jeho hodnoty bude používat samostatný stupeň vrcholu. Na základě výsledku můžeme také usoudit, že čím větší hodnotu obou parametru zvolíme, tím déle se hodnoty konduktance budou držet v nízkých hodnotách. O to více prudce však poté konduktance poroste nahoru, a kolem 85 % se všechny křivky začínají vyrovnávat. Vliv těchto parametrů lze vidět i na tomto grafu 22. Jelikož se jedná o konduktanci, kde nižší hodnoty jsou pro nás lepšími, můžeme usoudit, že naprosto nevhodné jsou pro nás hodnoty rovny 2 a 3 (dalo by se sice uvažovat o iteraci rovno 3 a počtu kroků rovno 6), protože hodnoty konduktance jsou příliš vysoké.

Výsledné hodnoty modularity jsou si velice podobné s výsledky konduktance, s tím rozdílem, že hodnota modularity je žádána co nejvyšší. Tento graf 23 zobrazuje opět výsledky všech experimentů. Z důvodu nečitelnosti jsou z grafu vytvořeny tři následující grafy, které nám pomohou pochopit výsledky tohoto experimentu.

Na prvním grafu 24 lze opět vidět průběh toho, jak se modularita mění v závislosti na parametru k, na druhém 25 pozorujeme změnu v závislosti na parametru t.

Podle uvedených grafů si lze všimnout, že nejvyšší hodnoty je možné najít při zadání hodnoty pravděpodobnosti v intervalu od 45 % do 65 %, ve kterých modularita roste od nejnižší hodnoty. Jak rychle roste závisí na parametru k a t, čím vyšší jsou, tím rychleji vzrostou, protože se později dostanou za nulovou hodnotu. Po dosažení vrcholu modularita začíná lehce klesat a všechny křivky se stejně jako u konduktance vyrovnávají. Po větším zkoumáním je možné se na grafu 24 povšimnou zajímavého faktu. Čím méně kroků je zadáno, tím větší je výsledná modularita. Z grafu 25 lze usoudit, že počet iterací nemá na výslednou modularitu příliš velký vliv.

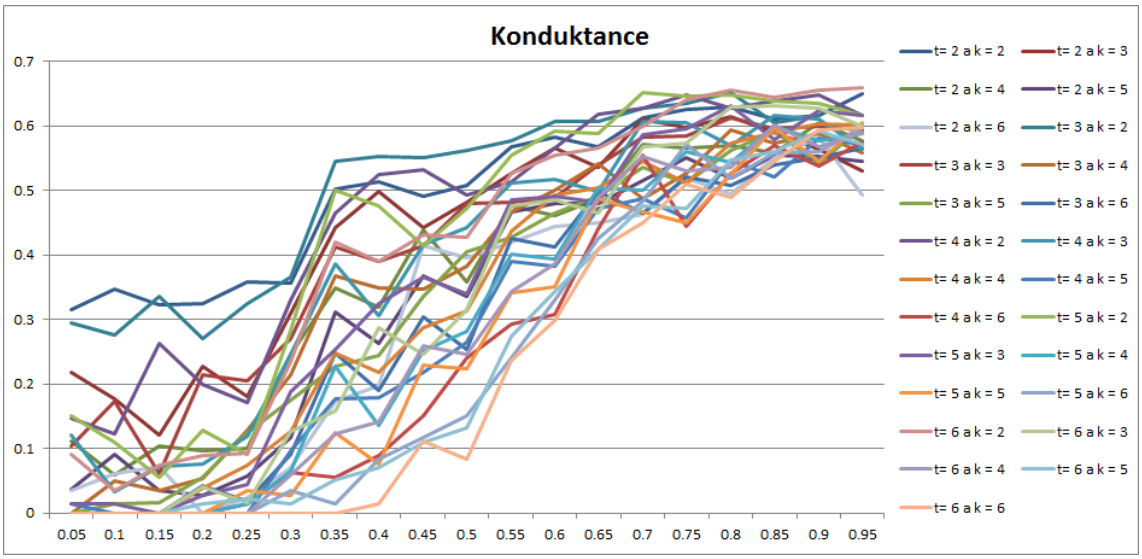
Obecně podle grafu 26 je možné usoudit, že čím nižší hodnoty parametru iterace a kroků jsou zvoleny, tím rychleji modularita začne růst a dosahuje vyšších hodnot.

Poslední parametr kvality, který chybí zanalyzovat je silhouette index. Výsledky experimentů lze nalézt v tomto grafu 27. Jeho výsledky byly také rozděleny do následujících grafů.

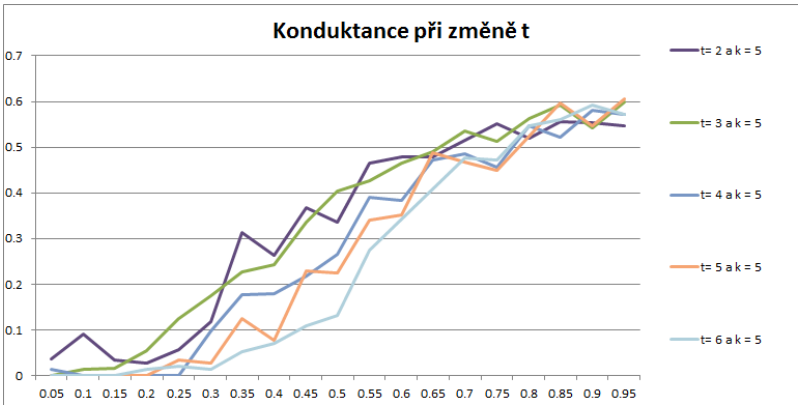
Jelikož je kvalita silhouette index úměrná s kvalitou konduktance, dalo se očekávat, že podle grafů 28, 29 a 30 dojdeme k závěru, že čím vyšší jsou zadané parametry kroků a iterace, tím vyšší bude výsledná hodnota silhouette index. Na grafu 30 můžeme vidět, že pro nižší vstupní parametry, se výstupní hodnota SI vůbec nerovná 1, a pokud ano, tak jen krátce. Poté začnou různě rychle klesat a kolem 90 % se začnou SI hodnoty rovnat.

Podle očekávání se nejlepší výsledek nenašel, je to kompromis toho, kterému kvalitativními algoritmu dáme přednost. Při nižších procentuálních hodnotách se konduktance blíží nule a SI zase 1, což ale kazí modularita, která se blíží v těchto případech také nule, protože v tomto případě je výsledný shluk často pouze jeden. Při zvyšování parametru pro minimální procentuální překryv, nám začíná vznikat více jak jeden výsledný shluk a modularita se z toho důvodu začíná zvyšovat. Konduktance a SI se však horší, protože vzniká

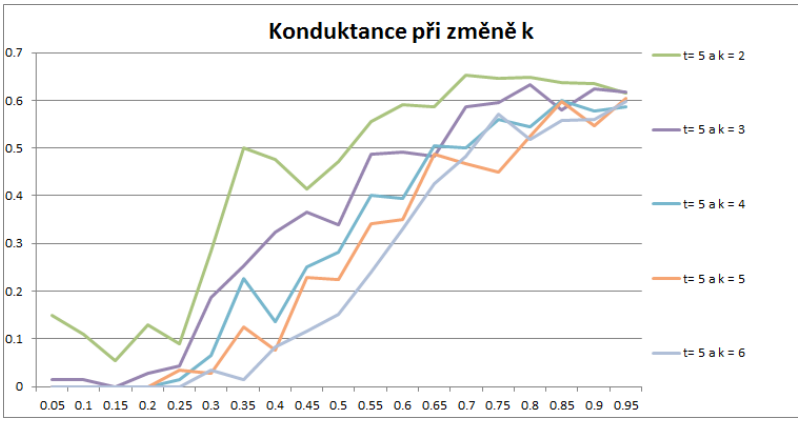
Obrázek 19: Výsledky testování konduktance



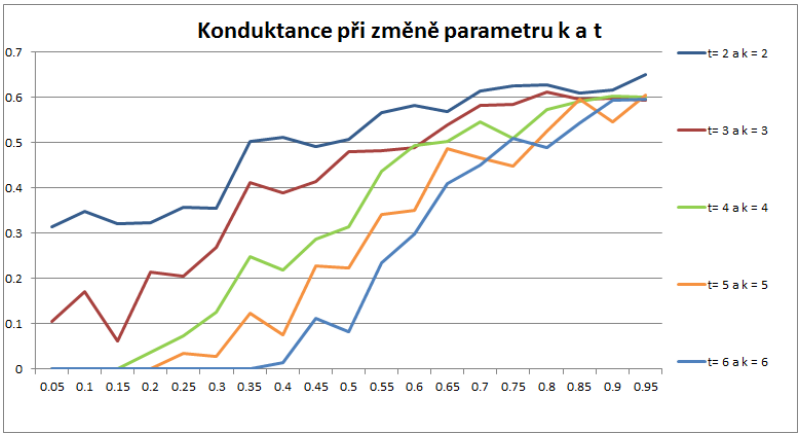
Obrázek 20: Konduktance při změně parametru t



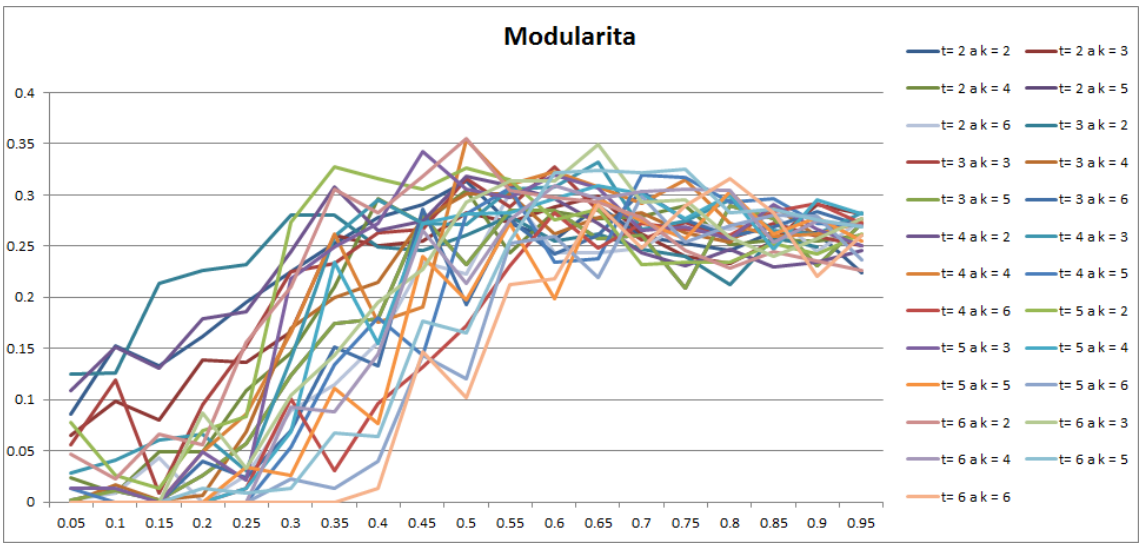
Obrázek 21: Konduktance při změně parametru k



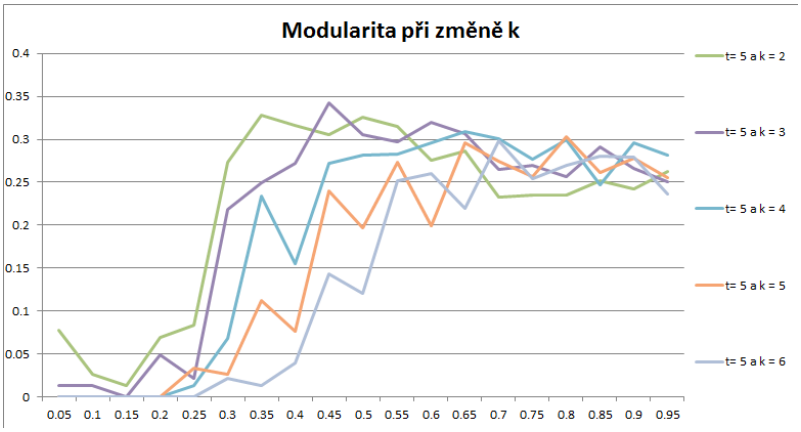
Obrázek 22: Konduktance při změně parametru k a t



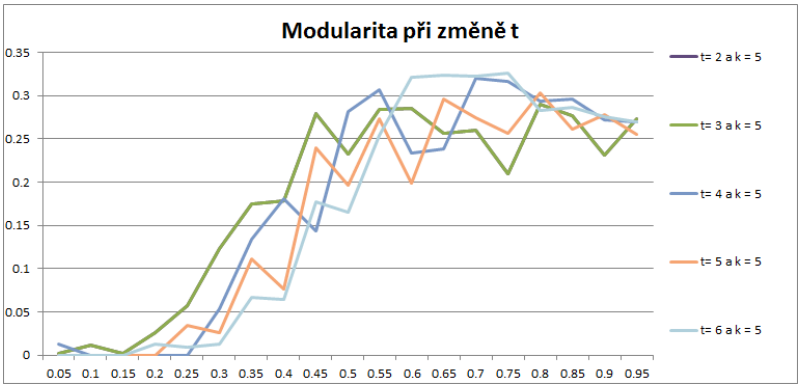
Obrázek 23: Výsledky testování modularity



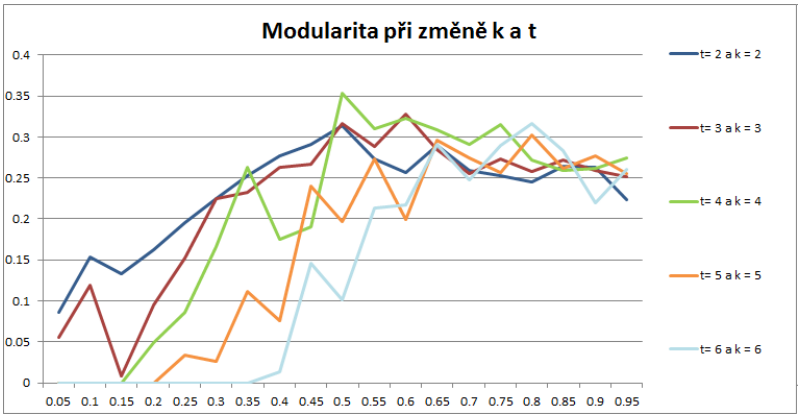
Obrázek 24: Modularita při změně parametru k



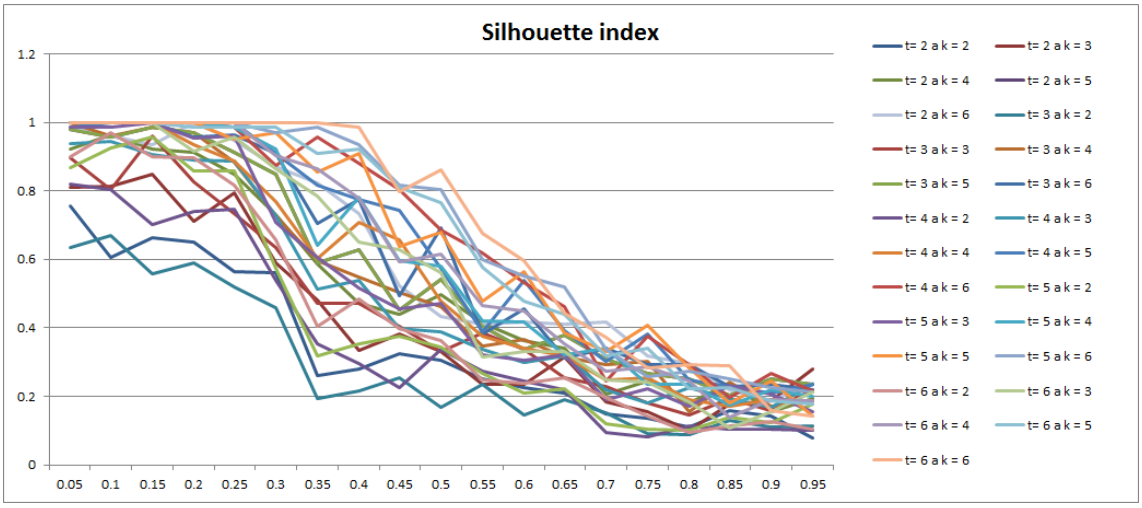
Obrázek 25: Modularita při změně parametru t



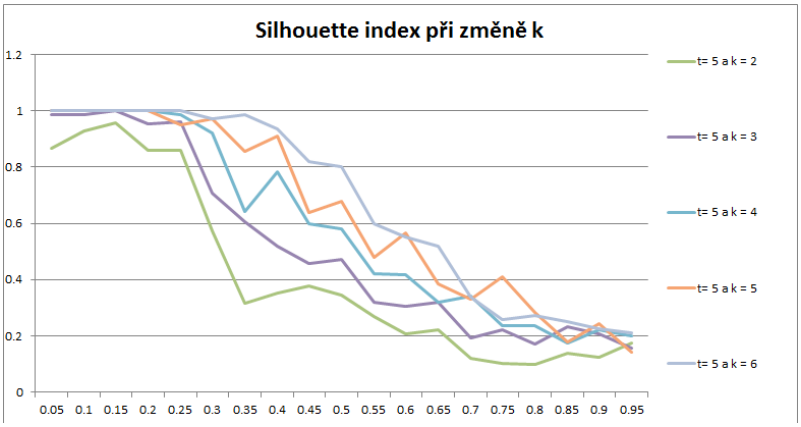
Obrázek 26: Modularita při změně parametru k a t



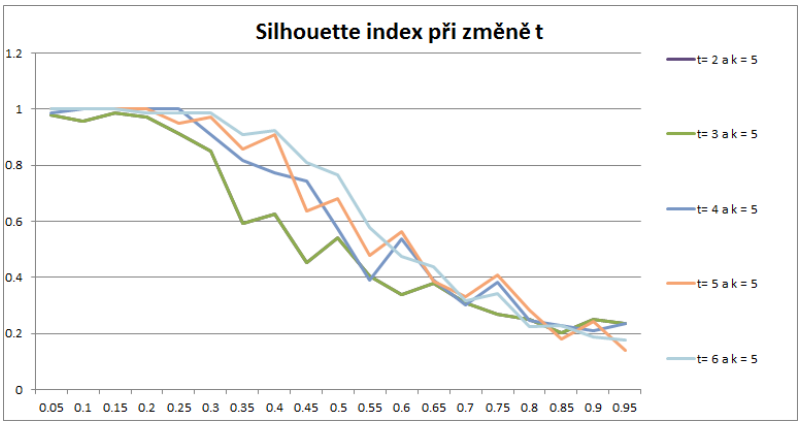
Obrázek 27: Výsledky testování silhouette index



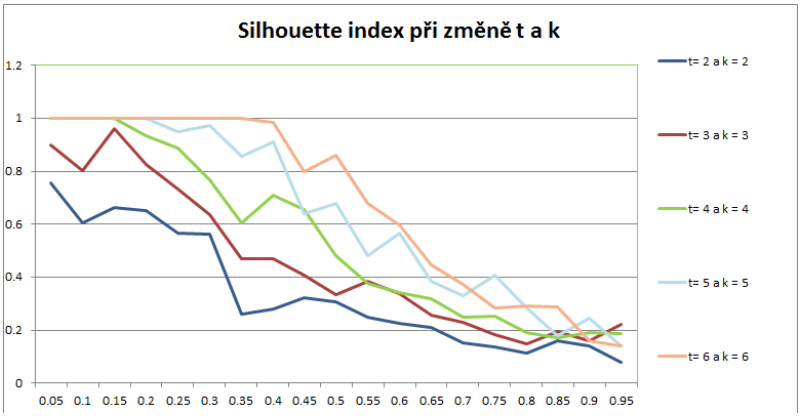
Obrázek 28: Silhouette index při změně parametru k



Obrázek 29: Silhouette index při změně parametru t



Obrázek 30: Silhouette index při změně parametru k a t



řez hran. Pokud bychom ale chtěli najít nejlepší výsledek s ohledem na všechny kvality, a museli bychom tedy udělat kompromis, pak by bylo zřejmě vhodné vybrat t a k rovno čtyřem, a pravděpodobnost zadat něco pod 50 % . Výsledek by vypadal takto: 13.

5.4 Porovnání sekvenčního RW algoritmu s RW využívající MPI

Poslední experiment, kterým se zabýváme je porovnání času sekvenčního algoritmu k algoritmům využívajícím MPI knihovnu. Porovnávají se také výsledky kvality algoritmů, z důvodu zjištění zda nedochází k nějakým výrazným změnám. Očekávaný výsledek tohoto experimentu je, že při využití malého grafu, by se časové zlepšení nemělo projevit. Je tedy použit graf Zachary karete klubu a také větší graf Enron. Graf Enron reprezentuje emailové propojení zaměstnanců stejnojmenné společnosti. Graf má 36692 uzlů a 183831 hran.

V rámci tohoto testování je využita, jako obvykle, hodnota 5 pro parametr počtu kroků a minimálního počtu sledů. Jako hodnotu parametru pravděpodobnosti jsme byla určena hodnota 50%, která na základě minulého experimentu spadá do rozmezí ideálního kompromisu mezi kvalitativními parametry. V tomto experimentu jsou dále porovnávány rozdíly času při využití více vypočetních uzlů, ale také samotné algoritmy mezi sebou. Na základě implementační části této práce si lze odvodit, že celkově porovnáváme čtyři kombinace. První je využití MPI knihovny, kde všechny výpočetní uzly vytvářejí jedno RW okolí, tento přístup lze využít jak pro variantu VS, tak NS. Obě tyto varianty lze použít také pro druhý způsob využívající MPI a to tam, kde každý výpočetní uzel vytváří vlastní RW okolí. První z těchto způsobů bude dále označován MPI a druhý MPI2.

Toto testování bylo provedeno na následujícím hardwaru. Byly využity dva servery Intel Xeon E5504 @ 2.00GHz, 8 GB RAM s operačním systémem Windows Server 2008R2. Na každém z těchto serveru běžely 4 virtuální servery, kde každý měl k dispozici čtyři procesorová jádra s 1024 MB RAM. Virtuální servery běžely na Windows Server 2012R2.

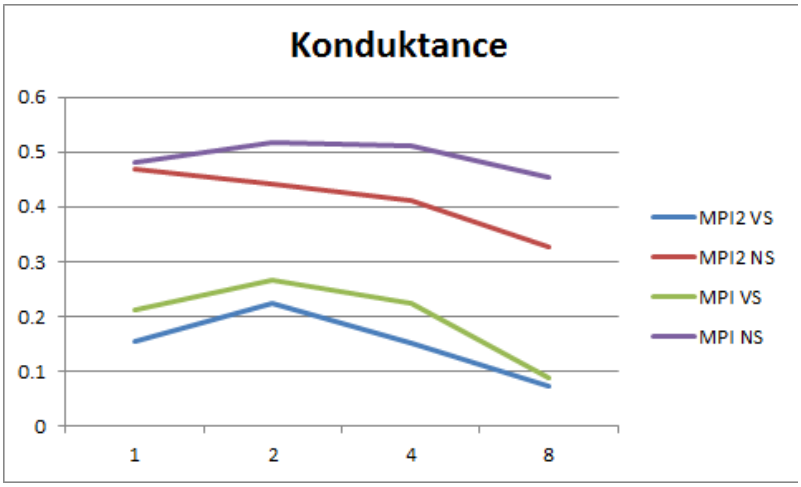
5.4.1 Výsledky MPI pro malý graf

Na výsledných datech konduktance (viz Obr. 31), modularity (viz Obr. 32) a SI (viz Obr. 33) si lze všimnout, že se žádné dramatické vychýlení nekonalo. Dokonce je patrné, že algoritmy NS drží při sobě, to stejné platí o algoritmech VS. Podle očekávání jsme se časového 34 zrychlení nedočkali, právě naopak. A to proto, že se na tak malém grafu nestačila síla použití více počítačů projevit.

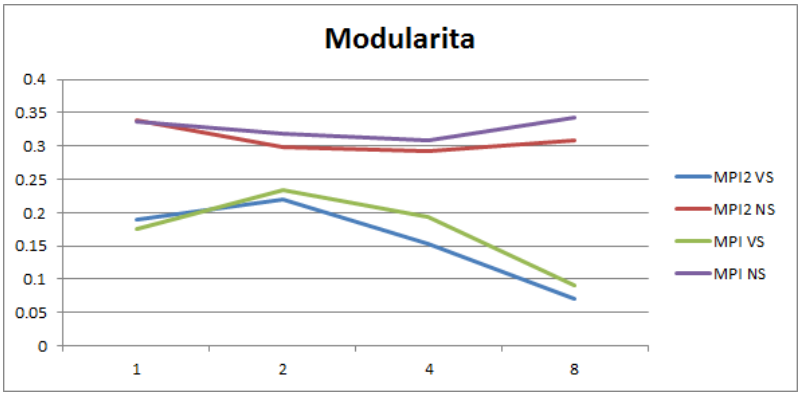
5.4.2 Výsledky MPI pro velký graf

V tomto případě se zrychlení výpočtu opravdu projevilo na algoritmech MPI2 35. Bohužel však není tak veliké, jak bychom si přáli. Největší důvodem tohoto malého zrychlení je, že výběr počátečních vrcholu z uzlů, které ještě nebyly přiděleny do žádného RW okolí je náhodný. Proto se stává, že v jednom odeslání počátečních vrcholů výpočetním uzlům je odeslán jeden vrchol v_1, který má stupeň 2 a další v_2, který má stupeň 300. V tomto případě výpočetní uzel s vrcholem v_1 skončí s vytvářením RW okolí daleko rychleji, než výpočetní uzel s vrcholem v_2. Řídicí uzel tedy čeká dokud neskončí poslední výpočetní uzel,

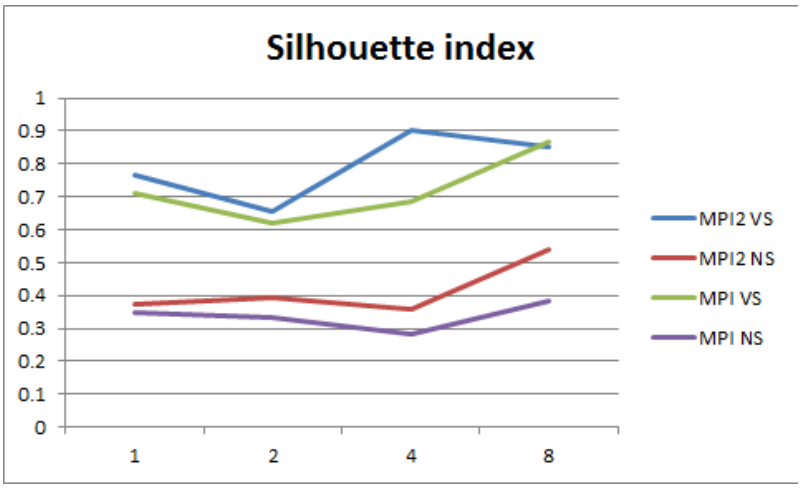
Obrázek 31: Konduktance při zvyšování počtu výpočetních uzlů na malém grafu



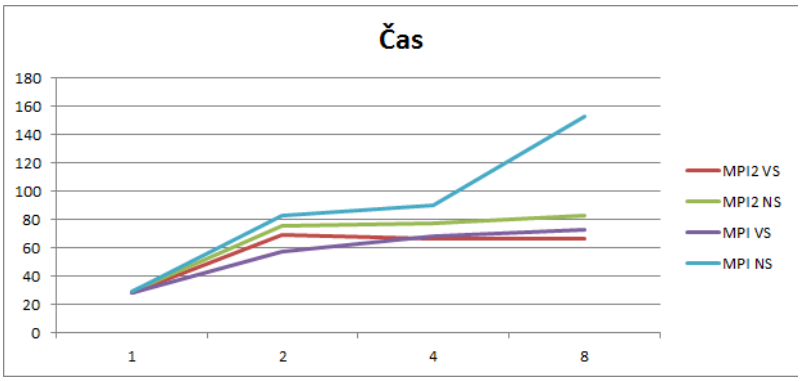
Obrázek 32: Modularita při zvyšování počtu výpočetních uzlů na malém grafu



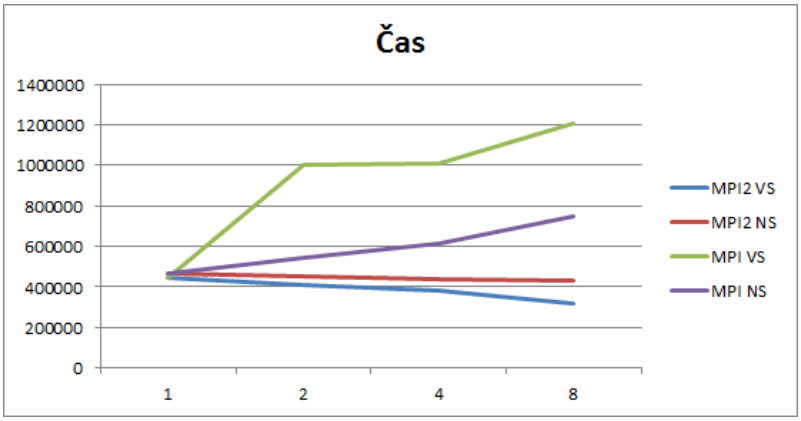
Obrázek 33: Silhouette index při zvyšování počtu výpočetních uzlů na malém grafu



Obrázek 34: Čas v milisekundách při zvyšování počtu výpočetních uzlů na malém grafu



Obrázek 35: Čas v milisekundách při zvyšování počtu výpočetních uzlů na velkém grafu

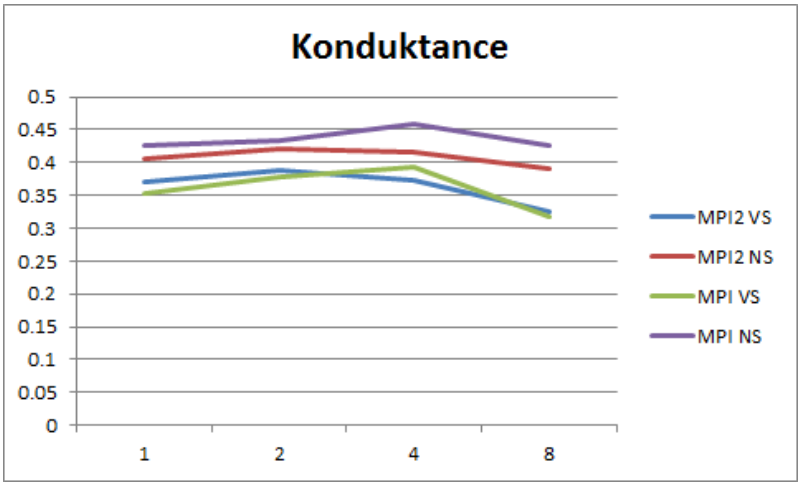


aby mohl přijmout výsledky a zpracovat je. V tomto případě už ale ostatní výpočetní uzly čekají na příjetí dalších počátečních uzlů. Takže v podstatě nedokážeme využít plnou sílu více výpočetních uzlů. Naproti tomu algoritmy MPI, kde se na vytváření každého RW okolí podílí každý výpočetní uzel, se zpomalují. Toto je způsobeno velkou komunikaci, která je mnohem častější než u algoritmu MPI2.

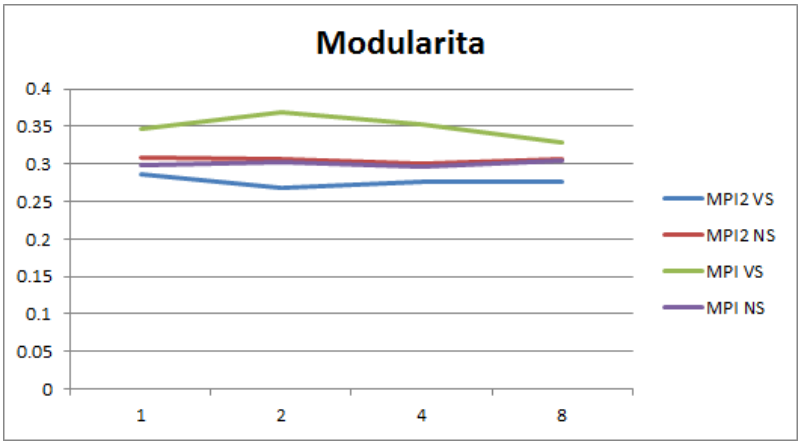
Kvalitativní výstupy na obrazcích 36, 36 a 38 ukazují, že také nedochází k výraznějším změnám.

Jedna z možností, která se nabízí pro urychlení MPI2, je vybírat počáteční vrcholy RW okolí na základě jejich stupně. Jednotlivé výpočetní uzly by pracovaly stejně dlouho a nedocházelo by tedy k čekání jednotlivých výpočetních uzlů na ostatní. Takové rozšíření je však samo o sobě natolik výpočetně náročné, že vzniká otázka, zda bz opravdu přispělo ke zrychlení celého algoritmu nebo naopak.

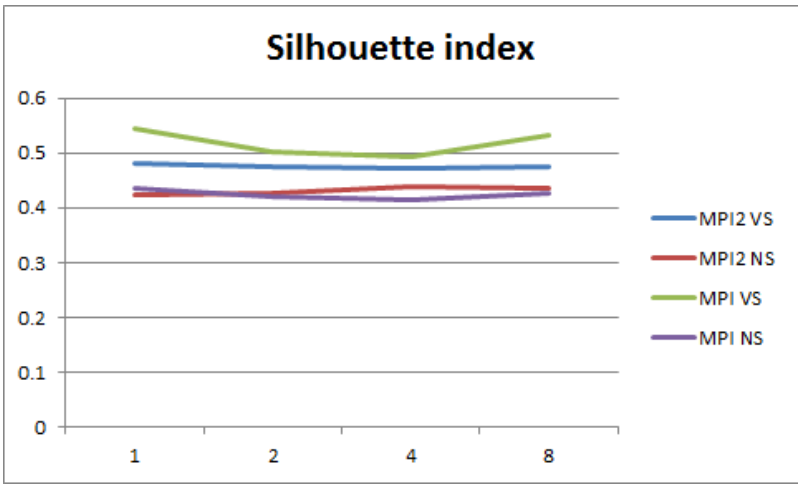
Obrázek 36: Konduktance při zvyšování počtu výpočetních uzlů na velkém grafu



Obrázek 37: Modularita při zvyšování počtu výpočetních uzlů na velkém grafu



Obrázek 38: Silhouette index při zvyšování počtu výpočetních uzlů na velkém grafu



6 Závěr

Již byly ukázány různé metody, které se zabývají shlukováním grafů, byl také rozebrán reálný program, který se v dnešní době používá. Po dokončení tohoto základního přehledu, byla věnována větší pozornost metodě *random walk*, a snaze najít její verzi, která by šla převést do varianty využívající MPI pro komunikaci mezi více výpočetními uzly.

Bohužel hledání této varianty RW dopadlo neúspěšně, proto nezbylo než přejít k návrhu vlastního intuitivního RW, který se jako každý vznikající algoritmus dočkal více variant. Po naivním přístupu, který pro nalezení každého RW okolí využívá každý výpočetní uzel označovaný jako MPI, došlo k přesunu k verzi MPI2, která za účelem snížení komunikace mezi uzly využívá k vytvoření RW okolí pouze jeden uzel.

Samotný způsob vybírání jednotlivých vrcholů do RW okolí není pouze jeden. První z nich využívá výskyty ve sledech VS, a druhý je odvozený od algoritmus *neighbour similarity* NS.

Kromě očekávaného výstupního souboru s výslednými shluky, byly vysvětleny a implementovány kvalitativní parametry, které určují jak kvalitní shluky byly v grafu nalezeny. Těmito kvalitativními parametry jsou modularita, konduktance a silhouette index.

Program lze ovládat pomocí několika argumentů, které se zadávají v konzolovém řádku. Hodnoty těchto argumentů přímo ovlivní samotný výsledek. Nejpodstatnějším z nich je argument *-p*, který určuje, jak moc se RW okolí musí překrývat, aby mohl byt označen za jeden větší RW okolí. Čím nižší má tento parametr hodnotu, tím méně výsledných shluku vznikne, jelikož se vytvoří větší shluky. Naopak čím vyšší je tato hodnota, tím těžší je pro RW okolí sjednotit se, z tohoto důvodu vznikne mnoho malých shluků. Dále je možné nastavit počet kroků v jednotlivých sledech a také minimální množství vytvořených sledů pro každý RW okolí. Velice prospěšný může být také parametr *-a*, který urychlí samotné načítání vstupního grafu při využití více výpočetních uzlů, protože si každý uzel načte graf sám ze sdílené složky na síti.

Provádění experimentů se skládalo ze čtyř částí, v první z nich se ukázaly samotné ukázky rozdělených grafů, a bylo zjištěno, že algoritmus je nedeterministický, a to kvůli náhodnosti při vytváření RW okolí a následném slučování. V pořadí druhý test se zaměřoval na porovnání verzí VS a NS. V tomto testu VS dopadl lépe pro kvalitativní parametry SI a konduktanci, ale NS dopadl lépe pro NS. Také bylo zjištěno, že výsledky jsou vždy o kompromisu mezi SI, konduktancí oproti modularitě. V následujícím testu byla tato doměnka potvrzena, protože byly porovnány výsledky při větším rozmezí zadaných parametrů. Ideální kompromis mezi kvalitativními parametry vyšel tak, že počet kroků a sledů je roven 4, a hodnota parametru *-p* pod 50%. V posledním testu, kde byl porovnán sekvenční RW algoritmus s paralelním RW bylo zjišteno, že při využití algoritmu MPI2 VS, bylo dosáženo největšího časového zrychlení.

Na základě těchto výsledku lze prohlásit, že je možné tento program úspěšně aplikovat na shlukování grafů s ohledem na časovou složitost. Jsou k dispozici vstupní parametry, které dovolí ovlivnit výsledek shlukování.

7 Reference

[1] Jakub Čech, *Abstraktní datový typ graf*, preprint (2006), dostupné zde `http://www.kiv.zcu.cz/~konopik/sem/cech/`.

[2] Lukáš Jirovský, *Teorie grafů*, preprint (2010), dostupné zde `http://teorie-grafu.cz/`.

[3] U. Elsner, *Graph partitioning - a survey*, MONARCH Dokumenten und Publikationsservice Germany, 1-58, 2005.

[4] L.-T.Liu, M.-T. Kuo, S.-C. Huang and C.-K.Cheng, *A gradient method on the initial partition of Fiduccia Mattheyses alghoritm*, In IEEE/ACM Int.Conf on Computer Aided Designed, pages 229-234. IEEE/ACM, Nov. 1995.

[5] B. Hendrickson, R. Leland, *An empirical study of static load balancing alghoriths*, In Proceeding of the Scalabe Hight Performance Computer Conference, pages 682 685, IEEE, 1994.

[6] A. Pothen, *Graph partitioning algorithms with applications to scientific computing*, In D. E. Keyes, A. H. Sameh, and V. Venkatakrishnan, editors, Parallel Numerical Algorithms. Kluwer Academic Press, 1995.

[7] R. Van Driessche, *Algorithms for static and dynamic Load Balancing on Parallel Computers*, PhD thesis, KU Leuven, Nov. 1995.

[8] G. L. Miler, S-H Teng, W. Turston a S.A. Vavasis. *Geometric separators for finite element meshes*, PSIAM Journal on Scietific Computing, 19(2):364-386, Mar. 1998.

[9] Y. Saad. *Iteractive methods for sparce linear system*, PWS Publishing Compay, Boston, MAS, 1996.

[10] C.M.Fiduccia and R. M. Mattheyses. *A linear time heuristic for improving network partitons*, Technical Report 82CRD130, General Electric Co, Corporate Research and Development Center, Schenectaty, MY, May 1982.

[11] B. Hendrickson and R. Leland. *A multilevel algorithm for partitioning graphs*, Technical Report SAND93-1301, Sandia National Laboratories, 1993. appeared in Proc. Supercomputing 95.

[12] G. Karypis and V. Kumar. *A fast and high quality multilevel sheme for partitioning irregular graphs*, SIAM Journal on Sceintific Computing, 20(1):359-392, 1999.

[13] R. Preis and R. Dieckmann. *The PARTY Partitioning*, Library User Guide Version 1.1. SFB 376, Univ Paderborn.

[14] F. R. K. Chung. *Spectral Graph Theory, Regional Conference Series in Mathematics*, 92:0160-7642, 1994/

[15] B. Mohar. *The Laplacian spectrum of graphs*, Preprint Series Dept. Math. University E. K. Ljubljana 26, 261, 1988.

[16] Pavla Kabelikova, *Graph Partitioning Using Spectral Methods*, preprint (2006), dostupné zde `http://www.fei.vsb.cz/k470/cs/theses/kabelikova_ing.pdf`.

[17] BALCÁREK, David. *Využití HPC při srovnání algoritmů pro výpočet vývoje tématu* [online]. 2012 [cit. 2014-03-06]. Diplomová práce. VŠB - Technická univerzita Ostrava, Fakulta elektrotechniky a informatiky. Vedoucí práce Jan Martinovič. Dostupné z: ¡http://theses.cz/id/ppwrdd/¿

[18] Gregor Dougles and Martin Benjamin, *MPI.NET Tutorial in C#*, preprint (2008), dostupné zde `http://www.osl.iu.edu/research/mpi.net/documentation/MPI.NET%20Tutorial.doc`.

[19] Ivan Zelinka, *Biologicky inspirované výpočty* , preprint (2008), dostupné zde `http://arg.vsb.cz/data/Vyuka/BIV-AUI.pdf`.

[20] Xin Sui, Donald Nguyen, Martin Burtscher, Keshav Pingali, *Parallel Graph Partitioning on Multicore Architectures*, preprint (2011), dostupné zde `https://www.cs.utexas.edu/~xinsui/publications/lcpc10_sui.pdf`.

[21] Kirk Schloegel, George Karypis, and Vipin Kumar, *Parallel static and dynamic multi-constraint graph partitioning*, preprint (2002), dostupné zde `http://glaros.dtc.umn.edu/gkhome/fetch/papers/pmcCCPE02.pdf`.

[22] A. George. *Nested dissection of a regular finite element mesh.*, SIAM Journal on Numerical Analysis, 10(2):345-363, 1973.

[23] U, Delling D, Gaertler M, Görke R, Hoefer M, Nikolski Z, Wagned D. On *Modularity clustering.* IEEE Transactions on Knowledge and Data Engineering, 2008

[24] Bingjing Cai, Ghaiying Wang, Huiru Zheng and Hui Wang, *An Improved Random Walk Based Clustering Algorithm for Community* IEEE International Conference on, 2011

[25] Pasi Fränti, *Cluster validation*, preprint (2014), dostupné zde `http://cs.uef.fi/pages/franti/cluster/Clustering-Part3.ppt`.

[26] Harel D, Koren Y. *On clustering using random walks* Proceedings of the conference on the Foundations of Software Technology and Theoretical Computer Science; Lecture Notes in Computer Science, 2245:18–41, 2001.

[27] Monte R. *The Random Walk For Dummies* MIT Undergraduate Journal of Mathematics, 2005.

[28] Charles M, Snell J. *Introduction to probability.* 2nd rev. ed. /. Providence, RI: American Mathematical Society, c1997, xi, 510 p. ISBN 08-218-0749-8, 2006